Getting the Most Out of Functional Coverage Tips and Techniques

Stephen J. D'Onofrio
Ambar Sarkar

Paradigm Works

stephen.donofrio@paradigm-works.com
ambar.sarkar@paradigm-works.com

ABSTRACT

Functional coverage plays an important role in constraint-random functional verification efforts. However, without proper upfront planning, verification teams often struggle to use functional coverage effectively. This paper details how we used functional coverage from the verification-planning stage to the functional verification closing phase. Functional verification planning, coverage instrumentation, and functional coverage reporting/analyzing are all tedious manual processes. This paper discusses methodologies that we have used to connect and track the coverage plan to the regression results. The paper also includes topics such as functional coverage planning, test ranking, and instrumentation guidelines.

Table of Contents

List of Figures

List of Tables

1 Introduction

Writing specialized test cases to verify each feature inside a design is not adequate for functional verification for many of today's increasingly complex integrated chip designs. Due to consumers insatiable demands for new product features coupled with breakthroughs in silicon capacity and synthesis technology, verification teams are confronted with testing increasing complex designs[i]. Many of these designs include integrated microprocessors (often referred to as System on Chip or SoC designs) and complicated new standardized protocols such as PCI Express, SPI, and AMBA. As a result, the combination of design features, configurations and interfaces produces an enormous verification space. For each additional features that a design contains, it often results in an exponentially growth of specialized tests. This is because design features often interact with other features and the verification team needs to check many of these permutations.

Over the past decade, a majority of verification teams have adopted constraint-random (sometimes-referred to as directed-random) testing methodology to verify their designs. Most verification experts agree that constraint-random verification methodology helps maximize testing the of the design's verification space with the least amount of work over the long run. This methodology relies on intelligent randomness to setup and drive traffic into the design and verify that design features function properly. Using a random testbench methodology entails more upfront work for developing random generators, automated checkers and functional coverage points but potentially saves more time on the backend by verifying random combinations of features automatically compared to developing a directed test suite for each scenario. The figure (Figure 1) below depicts the differences between random test and directed test development.

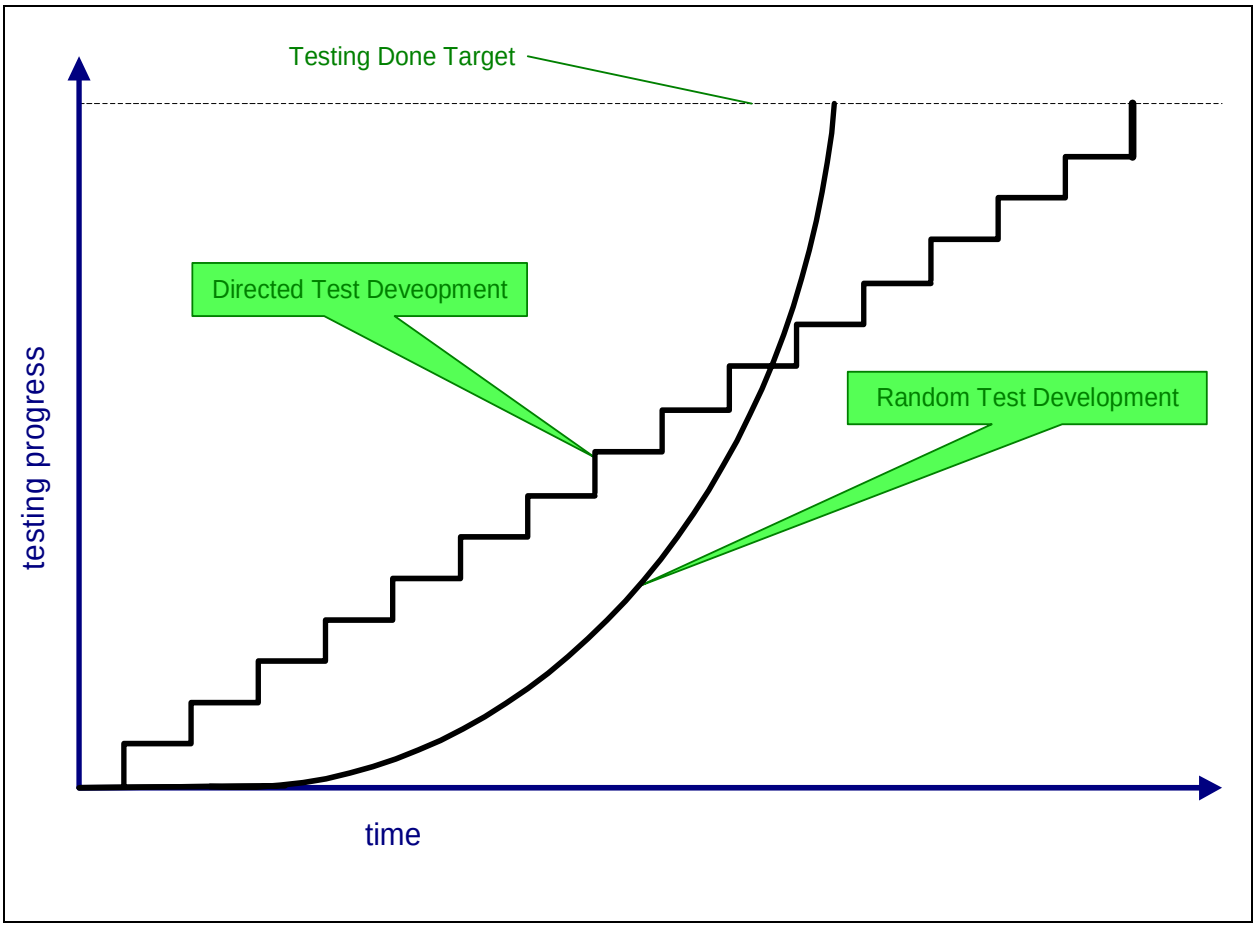


Figure 1 Verification Progress vs. Directed and Random Verification Methodology

Random testbench methodology involves three key components:

1. random generators
2. automated monitors/checkers/scoreboards
3. functional coverage

Random generators are in charge of setting up the design with sensible random configurations and random traffic. Constraints act as knobs in the testbench which control the generator's randomness. Ultimately, individual tests may extend the constraints to control the distribution and kinds of scenarios produced by the generators. The testbench also includes monitors/checkers/scoreboards that verify the functional correctness of design – this is the main purpose for us testing the design in the first place. These monitors/checkers/scoreboards must be robust in order to handle all the randomness that the generators present to them. Finally, *functional coverage* metrics identify which design features the testbench's randomness exercised.

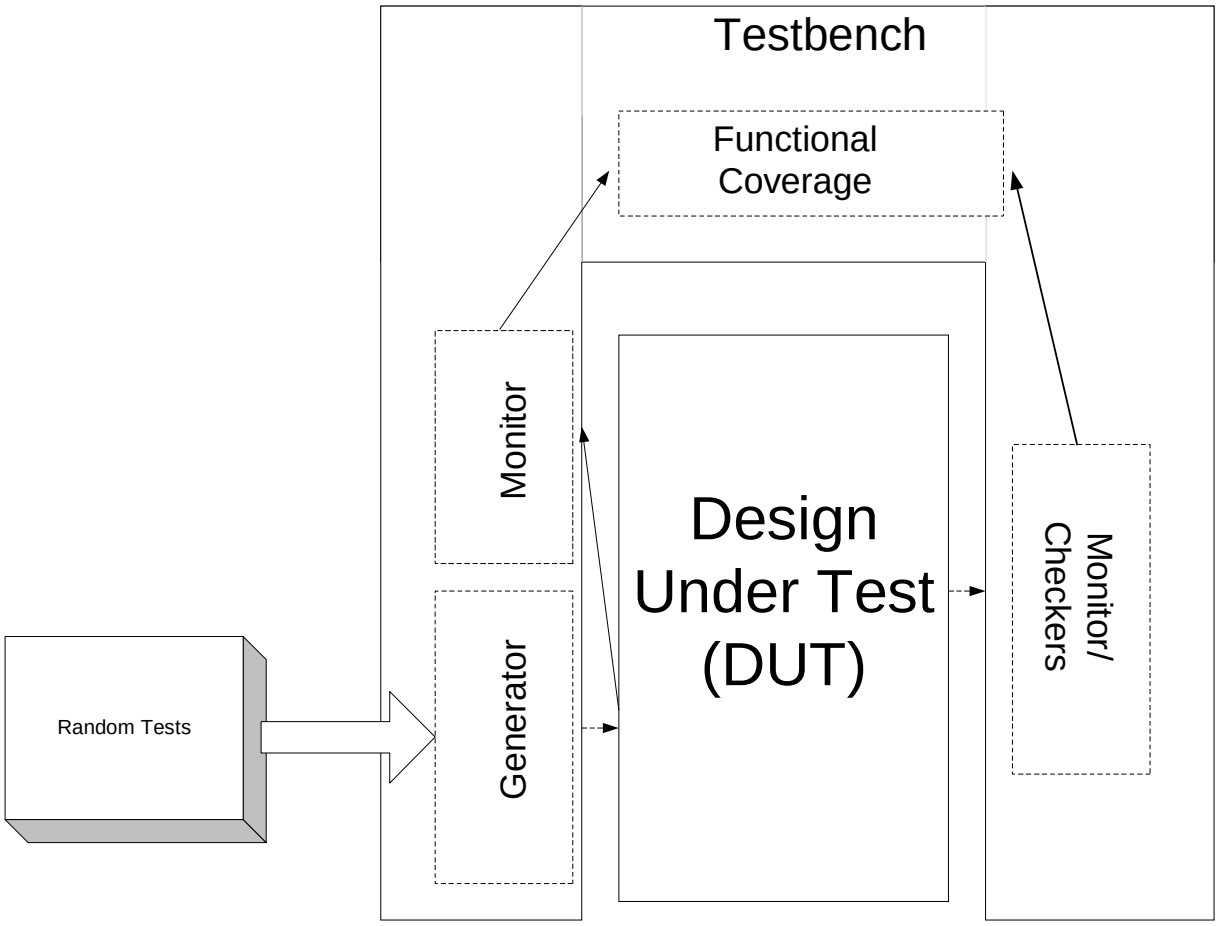


Figure 2 Random Testbench

Unfortunately, many verification efforts fail to utilize *functional coverage* or use *functional coverage* in an ad hoc manner without following any processes. Using *functional coverage* offers many benefits such as insight to functional verification progress, raises design confidence levels, and other points discussed later in this paper. Conversely, there is no argument that it takes more effort to incorporate functional coverage into the verification effort. Overall, the verification team's confidence increases by using functional coverage metrics to prove that the verification suite exercised and checked some fixed target of the design's verification space – the main objective for verifying the design in the first place. This paper discusses processes that we have successfully used in several projects and tools that we developed to help mitigate the amount of effort and optimize the usage of functional coverage.

2 Why use Functional Coverage?

2.1 What is Functional Coverage?

In the simplest terms, a *functional coverage point* merely updates a bin counter in a *functional coverage database* when a specific event occurs during a simulation. Verification engineers manually identify design features and then code them up as *functional coverage points.* Following a test or more typically a regression run, the verification team may generate *functional coverage*

reports from the functional coverage database. The *functional coverage reports* provides the verification team with *feedback* about which design features the testbench randomly exercised.

Functional coverage metrics helps the verification team gauge functional verification progress. Usually, the verification team analyses the functional coverage database's functional coverage metrics after a regression or simulation to determine whether a set of features was tested or not. Verification engineers often refer to these untested areas as *functional coverage holes*.

Functional coverage metrics do *not* necessarily imply that their associated features are bug free. Instead, *functional coverage* merely indicates that the verification environment *exercised* some design feature but not necessarily checked the integrity of the feature. Remember that it is the main goal of the verification effort to make sure the design is bug free and not to simply stimulate the design. Latter in this paper we discuss implementation techniques that help ensure that some functional coverage metrics may correlate with a properly checked feature.

There are three types of functional coverage points:

1. *item functional coverage point*
2. *cross functional coverage point*
3. *transitional functional coverage point*

Item functional coverage points capture information about scalar values. For example, when a design asserts a parity error interrupt, the verification environment needs to detect and check the behavior and then update a "crc error detected" coverage point. Another item functional coverage point may capture the kinds of packets passing through the design such as DLLP or TLP. The following table identifies these coverage points.

Coverage Point	Coverage Point Kind	Ranges/Bins
packet_kind	Item	DLLP TLP
crc_error	Item	CRC_ERROR, NO_CRC_ERROR

Table 1 Item Functional Coverage Points

Most times, it is also interesting to know if the randomness exercised two or more interesting combinations. *Cross-functional coverage points* collect combinations of *item coverage*. For example, a *cross-functional coverage point* gathers statistics about all the permutations of packets kinds such as DLLP packets and TLP packet with and the CRC error condition.

Coverage Point	Coverage Point Kind	Ranges/Bins
packet_kind	Item	DLLP, TLP
crc_error	Item	CRC_ERROR, NO_CRC_ERROR
Cross crc_error	Cross	DLLP with CRC_ERROR, TLP with NO_CRC_ERROR,

and crc_kind		DLLP with CRC_ERROR, TLP with NO_CRC_ERROR

Table 2 Cross-Functional Coverage Point

Occasionally, it is interesting to ensure that coverage items transition or sequence some pattern. *Transitional functional coverage* points capture sequences. For example, a *transitional functional coverage point* has the capacity to captures metrics about if a DLLP packets followed by one TLP packet.

Coverage Point	Coverage Point Kind	Ranges/Bins
packet_kind	Item	DLLP, TLP
crc_error	Item	CRC_ERROR, NO_CRC_ERROR
Cross crc_error and packet_kind	Cross	DLLP with CRC_ERROR, TLP with NO_CRC_ERROR, DLLP with CRC_ERROR, TLP with NO_CRC_ERROR
Transitional packet_kind	Transitional	DLLP->TLP, TLP->DLLP

Table 3 Transitional-Functional Coverage Point

HVL languages such as System Verilog, Vera, and Specman e provide functional coverage database features and coding constructs. Nonetheless, some verification teams build their own in-house coverage libraries using languages such as C/C++ most likely due to cost, legacy and resistance to change reasons.

The verification team needs to identify which design features need functional coverage points, determine functional coverage goals, develop coding guidelines used to instrument the functional coverage points, and determine a methodology for running tests and gathering functional coverage metrics to determine if their functional coverage goals have been met. The amount of functional coverage data can become overwhelming to manage. It is critical that a verification effort includes a verification plan that accounts for all these points early in the process.

2.2 Why not use only Code Coverage?

Code coverage metrics capture low-level information about the RTL. Code coverage information includes statistics about RTL line execution, branch conditions, pins toggling, and FSM state traversing. Generating code coverage reports does not require any upfront coverage-point identification process or coding. Although, code coverage does requires in-depth manual analysis of the reports. Why does a verification effort not solely use code coverage to determine doneness?

Code coverage is by definition limited to the design code it covers. Therefore, if a designer failed to implement a feature in a design, the tool may still report 100% coverage. For example, consider the case where the feature of interest is odd parity generation. If the designer forgot to

add any code that generates parity, code coverage may still result in a false positive of 100% statement/line coverage. If the verification team somehow let parity checking fall through the cracks, no one would be the wiser.

Another example, it may be impossible to determine if a design received certain stimulus inputs even if code coverage analysis is used. For example, it would be impossible to determine if a PCI Express design received two FC Init 2 DLLPs immediately following by a TLP packet.

Determining functional coverage completeness using code-coverage metrics is a tedious process. Functional coverage, unlike code coverage points, is at the hardware specification level of abstraction. Verification engineers work at the hardware specification level of abstraction so functional coverage is a natural fit for them. The verification process ultimately proves that the design features match the hardware specification descriptions. On the other hand, code coverage metrics are at the RTL implementation level. In order to determine if certain design feature was exercised you need to have very intimate knowledge of the design RTL implementation details using code coverage. Additionally, it is risky if verification engineers bias their testing because of their knowledge of the RTL implementation details, especially early on in the verification process.

Intrusively modifying RTL code to add pragmas and/or developing a script that automatically sift through code coverage results to determine if code coverage exercised a feature after a regression(s) run is a non-practical, risky and tedious task, especially for verification engineers. It is sometimes a risky proposition to modify RTL files nearing the tape out phase in order to add pragmas. Furthermore, it is not very scalable because if the design changes then the pragmas and/or script may need updating. Again, the code coverage metrics are at the RTL implementation level. Alternatively, analyzing functional coverage points after a regression is a more practical and straightforward task at the hardware specification level and is non-intrusive to the RTL code. Functional coverage points are controllable from a regression level. Users can easily control functional coverage attributes including weights, goals, and minimum number of hits. Additionally, verification teams may develop scripts or use third party EDA tools that have capabilities for filtering and mapping functional coverage point database metrics to hardware design features as discussed latter in this paper.

2.3 Some of the barriers to using functional coverage

Often we find that management and design teams resist the notion of functional coverage because of its novelty or their misinterpretation of what functional coverage is. Additionally, functional coverage requires more upfront work for functional coverage point identification, more code development time for implementing functional coverage points, and analysis of the functional coverage results.

Functional coverage is a relatively new concept for mainstream functional verification. It is somewhat simpler for the teams to grasp the concept of developing test cases to verify a design feature rather than identifying and hitting functional coverage points. For example, a design may include a feature that drops ATM cells if the length is not equal to 53 bytes and this causes a status bit and possibly an interrupt to assert depending on its mask field. The verification team

may gain confidence that a design feature is operating properly by developing the following tests cases:

(1) Send in Packet with Size equal to 53 and verify that it is not dropped with mask set plus make sure status is clear and interrupt does not occur
(2) Send in Packet with Size equal to 53 and verify that it is not dropped with mask not set plus make sure status is clear and interrupt does not occur
(3) Send a Packet with Size != 53 and verify dropped with mask set and interrupt is not asserted and status is set
(4) Send a Packet with Size != 53 and verify dropped with mask clear and interrupt is asserted and status is set

Alternatively, the verification team may gain confidents about the ATM drop feature by identifying the functional coverage points illustrated below.

Coverage Point	Coverage Point Kind	Ranges/Bins
ATM_Packet_Status	Item	ATM_PACKET_DROP, ATM_PACKET_NOT_DROP
ATM_Drop_Mask	Item	ATM_DROP_STATUS_SET, ATM_DROP_STATUS_CLEAR
Cross ATM_Packet_Status and ATM_Drop_Mask	Cross	ATM_PACKET_DROP with ATM_DROP_STATUS_SET, ATM_PACKET_NOT_DROP with ATM_DROP_STATUS_SET, ATM_PACKET_DROP with ATM_DROP_STATUS_CLEAR, ATM_PACKET_NOT_DROP with ATM_DROP_STATUS_CLEAR

It is obviously more intuitive to understand how test cases prove that a feature is tested compared to a functional coverage points. One must have a basic understanding of functional coverage points concepts and a basic understanding of a directed random testing environment in order to comprehend how functional coverage points prove the a design feature.

Folks must understand that the random environment uses constrainable properties to generate ATM packets with size equal and not equal to 53 and the ATM Drop mask is set and cleared randomly. The environment sets up some default random distribution, for example 75% distribution of ATM packets with size equal to 53 and 50% distribution of the ATM Mask being set. The monitors inside the verification environment update a functional coverage database bin when ATM packets are dropped and interrupts are processed. The verification environment is capable of generating post regression functional coverage reports so that the team can detect if the functional coverage points above were hit or not. The team needs to understand the verification environment and functional coverage concepts from a high-level perspective without becoming too caught-up in fine details.

Understanding random testbench test cases is another area of confusion for teams new to functional verification. The question of how do you write tests to verify features in a random test environment is often asked. Ideally, inside a random verification environment there is no need to develop a special test case needed to verify the ATM Drop feature. Instead, in a regression a single *random* test is run multiple times using different seeds in order to hit all the functional coverage points. In some cases, say a design performance test that explicitly verifies throughput

with *good* packets, it may be necessary to create a special test case that overrides the ATM Drop distribution to 0%. In other hypothetical cases, say that after testing with many seeds and not hitting the cross condition "ATM_PACKET_NOT_DROP with ATM_DROP_STATUS_CLEAR" then the verification team may opt to put together a new test case that intentionally constrains ATM packets size with size 53 and the ATM Drop mask to set.

The team must also realize the benefits of directed random testing over writing specialized test cases as shown in Figure 1. This figure shows that when a design reaches a certain degree of complexity it takes more effort to continue to develop directed test cases compared to developing the directed random verification environment and running random tests multiple times.

The design/architecture teams should be involved in reviewing the functional coverage identification process as described latter in this paper and understand the benefits of directed random testing. After educating and involving the team in this process, they usually start to become more receptive to the functional coverage approach. The management team's concerns are generally further eased when we use a tool or develop a script that can generate functional coverage progress reports from regression runs. These reports help the "management team" track progress.

3 Verification Planning

A verification plan may be a single document or a library of documents. Information such as functional coverage plans (sometimes referred to as functional coverage models), random testbench architecture, and doneness criteria are all included in the verification plan. The management and architecture teams should review appropriate aspects of the verification plan. Typically, most documents in a verification plan are between ten to forty pages. Be aware that we find that engineers are often hesitant to read and review large documents.

The verification plan should list all the components that the verification team needs to develop such as modules, monitors, sequence drivers, bfms and transactors. These components break up into three categories; passive, active, and reactive. The plan's descriptions should clearly describe how these components are to be reused and configured for unit levels and system level testbenches.

The entire verification team should participate in putting together the verification plan. A verification team needs a lead that ideally has a good degree of verification engineering experience. The verification lead should develop the high-level details such as the doneness criteria and unit to system-level integration methodology. The verification engineers assigned to test certain areas of the design should write the functional coverage plan and testbench description pertaining to that area. The lead verification engineer needs to review and make final decisions on all aspects of the verification plan.

3.1 Why write a Verification Plan

Today's SoC designs contain complex functionality that needs verification before entering the expensive silicon process. The verification environments often consist of multiple unit and system level testbenches. Netlist and/or accelerator simulations may also need to reuse some or all the

functional verification code. The team usually has tight deadlines and the team grows towards the tape out deadline. It is foolish to attempt to coordinate such a complex effort without a clearly defined plan that appropriate architectural and management team members agree on. A very important objective of the verification plan should be to identify a doneness criterion for the verification effort.

3.2 Doneness Criteria

The verification plan should clearly spell out a "doneness" criteria that is reviewed and agreed upon with the management team. For functional coverage, the doneness criteria should answer the questions what are the expected functional coverage grades and when functional coverage should be collected and merged. An example of a "doneness" criterion is shown below:

"Verification testing will finish only after:

(1) Verification plans signed off by the system architect
(2) All functional coverage plans have grades of 100%
(3) Functional Coverage metrics are collected against fully passing regressions. The functional coverage results from multiple regressions may be merged together assuming that the design has not changed.
(4) Netlist simulations run
(5) Statement/line code coverage is as close to 100% as possible with all lines that are not hit signed off by the designer and archived in the issue tracker
(6) FSM coverage as close to 100% as possible with all lines that are not hit signed off by the designer and archived in the issue tracker
(7) No bugs are reported for a 2 week period"

3.3 Functional Coverage Plan

Another key component of the verification plan is the functional coverage plans. Often, the testbench implementation deviates from the verification plan's description over time. However, for the functional coverage plans this is not true. These plans must stay *living* since they influence the level of testing in the "doneness" criteria.

A functional coverage plan identifies a target for the random test environment to exercise. Typically, a verification plan breaks up each unit level design, the system level, and common interfaces into individual functional coverage plan documents. We find it is best to further break up an individual functional coverage plan into a hierarchy of sections that match the design and/or hardware specification in some logically manner. Sometimes, it is possible to reuse functional coverage plans in multiple projects.

3.3.1 What Does a Functional Coverage Plan Look Like?

Typically, we find that verification teams usually writes their functional coverage plan documents using Adobe Frame, MS Word or a spreadsheet application. Below is a simplified example of a few functional coverage points for a PCI Express Functional Coverage Plan that uses MS Word. The example below illustrates the minimal amount of information that is necessary for a functional coverage plan.

The functional coverage plan's table below lists individual functional coverage points. Each functional coverage point has a unique name listed in the first column. The next column is the functional coverage point's property *kind*. A functional coverage kind is either *item, cross,* or *transitional*. The description and values columns are utmost important because they detail what the coverage point is and identify specific ranges or values for the functional coverage point. It is crucial that folks reviewing the functional coverage plan understand these columns.

The goal column specifies what the score the individual functional coverage point needs to have to reach functional coverage to goal. Usually, it makes sense for item functional coverage points to have goals of 100% in order to have confidence in the design – this is discretionary to the verification effort. Finally, the last column is the reference column. It is wise for verification plan to reference the hardware specifications or other documentation sources. The specification's version number should also be included. It is best to reference where information is rather than copying it into the verification plan. Often, a design change and becomes a maintenance headache to keep up with updating the verification plans. We feel it is best to have one copy of the design specification details – it does not belong in the verification plan.

(1) DLL Section

a. DLL Packets

Name	Kind	Description	Ranges/Bins	Goal	Reference
dllp_type	**ITEM**	**This coverage point captures all possible dllp packet kinds**	**ACK, NAK, PM_ENTER_L1, PM_ENTER_L23, PM_ACTIVE_STATE_REQUEST_L1, PM_REQUEST_ACK, VENDOR_SPECIFIC, INITFC1_P, INITFC1_NP, INITFC1_CPL, INITFC2_P, INITFC2_NP, INITFC2_CPL, UPDATEFC_P, UPDATEFC_NP, UPDATEFC_CPL, TLP_DLLP**	**100**	**Section 3.4.1 in PCI Express Base Specification 1.0**
dllp_kind	**ITEM**	**This coverage point captures the dllp category as either a Link Management Packet (DLLPs) or a Data Exchange (TLPs to/from PHY and TL Layer).**	**LINK_MANAGEMENT, DATA_EXCHANGE**	**100**	**Same as above**

Table 4 Functional Coverage Table Example 1

b. DLL Flow Control

Name	Kind	Description	Ranges/Bins	Goal	Reference
dllp_flow_control_kind	**ITEM**	**This coverage point captures and categorizes that all flow control triplet**	**FC_INIT1, FC_INIT2, FC_UPDATE**	**100**	**Section 3.5.2.1 in the PCI Express Base Specification 1.0**

		modes have been exercised.			
vc_id	**ITEM**	**This is the virtual channel of the FC Packet. NOTE: For XXX design there are 2 virtual channels**	**VC_ID_0, VC_ID_1**	**100**	**Same as above**
Cross dllp_flow_control_kind, vc_id	**CROSS**	**See descriptions above!**	**FC_INIT1/VC_ID_0, FC_INIT2/ VC_ID_0, RF_UPDATE/ VC_ID_0 FC_INIT1/VC_ID_1, FC_INIT2/ VC_ID_1, RF_UPDATE/ VC_ID_1**	**100**	**Same as above**

Table 5 Functional Coverage Table Example 2

3.3.2 Reviewing the functional coverage plan

The verification, system architecture, and design team should review and signed-off all of functional coverage plans. They need to review all the functional coverage points included in the functional coverage plan. Additionally, the verification lead must educate the architecture and design teams about functional coverage and random testing environments as discussed earlier in this paper. At the end of the meeting, all parties need to agree if the functional coverage plan's points are adequately for verifying all the design's features.

Functional coverage plans should also be broken up into sections that map to logical partitions of the design. It is not appropriate to invite all the team members from the verification, design and architecture teams to attend the reviews all the functional coverage plans. Use common sense and only invite appropriate team members. However, the verification lead should review all the functional coverage plans and attend all the functional coverage plan meetings.

When a verification effort does not include a signed-off functional coverage plan, the team has no clear idea of what the verification team is testing. Additionally, verification engineers may get tempted to reduce the rigidity of some functional coverage points during functional coverage closure. A policy that we use with our signed-off functional coverage plans is that if a functional coverage point needs to change, then the change needs approval and a tracking tool records the change.

3.3.3 When should one come up with the functional coverage plan

Ideally, early verification planning and sign-off is good because it gets the entire team looking at the verification effort and gives the verification team a clearer idea of how much work lays ahead. From this information, we can roughly determine how many assertions and monitors/checkers we need to build in the verification environment. The verification team needs to update testbench architecture documentation with new verification components discovered during the functional coverage planning process. Finally, the verification team can generate a preliminary schedule based on the functional coverage plans and testbench architecture.

The verification plan data also helps clarify the understanding of the design for the verification team and sometimes sparks up interesting architectural issues for the designers. Additionally, the sign-off meetings allows for a nice forum for discussing "design for verification" strategies. For example, it can greatly improve verification productivity if the designer can implement special design hooks for loading counters, writing to RO registers, generating parity errors, etc. The designer may also have interesting scenarios that they feel are risky. The functional coverage plans and/or testbench architecture documentation needs to capture all this information.

3.3.4 Iterative process for the functional coverage plan

Design architectures are most often a moving target during the functional verification effort, especially early on in a project. Occasionally early in the project, a design's architectures may greatly lack maturity and it is not wise at this time to have an early functional coverage plan "sign-off" meeting. In these cases, it is best for the verification team to write the functional coverage plan in an iterative fashion and wait until the architecture team feels the design is solid before setting up the functional coverage sign-off meeting. During the iterative process, it is a good idea to add action items in a tracking tool about incomplete features and design descriptions that are not clear.

During verification development, after the functional coverage plan is signed-off, we find that certain functional coverage points may need to change or be added because of a corner cases uncovered by a bug, initial misunderstandings of the specification, and/or design changes. A tracking tool should capture all functional coverage plan changes. Additionally, appropriate parties should approve the changes. Adding new functional coverage points to the plans compared to removal or changes to existing points does not usually need such scrutiny since they are expanding the coverage target.

3.3.5 Functional coverage point attribute planning

Generally, it makes most sense to set your functional coverage point attribute *goals* to 100%. However, occasionally it may be appropriate for a cross-functional coverage points to have a *goal* of less than 100%. This may be the case when hitting multiple conditions does not exercise any new logic or it may be too time consuming with low risk to hit less than 100 % of all the permutations. For example, in the past we tested areas of a design that utilized design IP. If the design IP is silicon proven, the code was not modified, and if the configuration setup matched that inside the proven design then we would relax the amount of functionality testing for the IP. Instead, the focus was to verify the connectivity of the design IP. In some cases, we relied on the IP feature testing to occur latter in software co-verification or hardware accelerator testing rather than functional simulations. The verification teams should capture information such as justifying the level of testing inside the functional coverage plan.

Functional coverage points include an *at least count* attribute that specifies the minimum number of times a bin should be hit for it to be considered covered. Generally, functional coverage points use the default value of 1 for the *at least count*. Occasionally, we have used the *at least count* attribute to prove that certain features are exercised more than once in a simulation run. For

example, with interrupt testing it is interesting to prove that an error status is set and cleared. By setting the error status functional coverage point's *at least count* attribute to 2 this proves that the interrupt was set, serviced and cleared scenario occurred twice. It may be necessary to add a new column for the *at least count* or at least mention the *at least count* setting if a functional coverage plan uses an *at least count* setting other than 1.

Functional coverage grades are calculated using a weighted sum of all the functional coverage points. Each functional coverage point has a *weight* attribute and it defaults to a value of 1. We have never been involved in a project that manipulates the *weight* attribute. However, the VMM Manual for System Verilog [ii] recommends (Recommendation 6.23) that the coverage groups setup a linear proportional functional coverage-point weight methodology. Additionally, the VMM Manual recommends (Recommendation 6.22) that if a coverage point is used in a cross coverage point than the coverage point should have a *weight* of 0 since it will artificially raise the grade calculation. The VMM manual states that the cross coverage goal is a truer representative goal. However, if a cross coverage point has *ignore* statements then the associated coverage points holes may not surface in very rare cases. Overall, the functional coverage plan may manipulation the *weight* attribute solely for accounting purposes.

One issue that we discovered is that verification engineers new to functional coverage often put together extremely large cross coverage definitions. Cross-coverage functional coverage points grow exponentially. More coverage bins may mean unnecessary testing. Functional coverage items variables with very large ranges often contribute to this problem. We usually find it sufficient for putting together ranges that hit the minimum, maximum, and some random value in the middle and do not cross an excessive amount of coverage items. The functional coverage plan should clearly describe all the coverage ranges.

3.3.6 Instance vs. Cumulative Goals

Coverage statistics can be gathered either on a cumulative or on a per-instance basis. It is common to have interfaces that are instantiated multiple times in a verification environment. An example is a design that includes two AMBA multiplexes or four TDM interfaces. Most often, we gather functional coverage on a per instance basis because we want to ensure that the random stimulus exercises each interface instance. On some occasions, if we have confidence in an interface we will relax the verification rigidity and use cumulative coverage. If a functional coverage point is instantiated multiple times then the functional coverage plan should identify if the coverage point is using cumulative or instance functional coverage.

3.3.7 Directed Test Cases for the Functional Coverage Plan

Although the norm is to develop very random tests in a random verification environment it sometimes it makes sense to develop some highly specific directed tests for simple operations and update a special scenario coverage point that indicates that the regression ran the test. For example, one of the first tests we usually develop is a register test that randomly parses through all the registers, verifies default values, randomly write data to read-write register values, and

proves resetting the registers. The functional coverage plan should include a detailed description of the test.

3.4 Verification Components and Functional Coverage

A verification plan should contain a list that identifies all the reused and new verification components verification IP utilized by the verification effort. The list should additionally include information about if the components were developed in-house or by a third party vendor. The verification plan should indicate how the testbench configures the devices in the verification environment and include references to the verification component's documentation. The verification lead should investigate all third party-IP to verify its integrity.

Most verification components (VIPs, or eVCs or uVCs) that we used in the past contain built-in functional coverage points. Although having preexisting functional coverage points available is nice, some of the functional coverage points may not be suitable for the verification environment and these coverage points must not be included in the functional coverage plan and resulting functional coverage grade. These functional coverage points may be unnecessary because they belong to some mode or configuration that is not appropriate for the design or simply uninteresting for the functional coverage effort. The functional coverage plan should only include functional coverage points that are interesting for the verification effort.

It is vital to calculate grades by including only the functional coverage points included in the functional coverage plan. In some verification efforts in the past, we used a third party EDA tool and in others, we developed our own scripts to gather grades on functional coverage points that are only included in our functional coverage plans. This allowed us to avoid calculating unused functional coverage points in the final grades.

3.5 What role does Code Coverage play with Functional Coverage?

Another important aspect of our verification plan is the "doneness" criteria for measuring code coverage. Code coverage complements functional coverage metrics by blessing that the functional coverage plan is adequate. If you have 100% functional coverage and low line coverage results then your functional coverage plan or the design specification is deficient. Our plans also included using code coverage to make sure our tests exercised all the design's state-machines and toggled all the top-level pins. In some cases, we were able to eliminate some functional coverage points because the FSM coverage gave us confidence that the code was exercised.

3.6 Challenges Writing the Verification Plan

A Hardware Specification is a primary source for design specific information used in writing Verification Plans. Hardware Specification documents describe how the design functions, which in turn is usually based on a marketing Product Requirement's Document (PRD). Design Engineers author the Hardware Specification. Often we find that Hardware Specifications do not fully describe all functionality of the design and tend to be inherently ambiguous. It is often difficult to understand how the design operates and decipher functional coverage point information from these documents alone. Typically, the Design Engineers explain missing information and confusing oddities on a white board or in some other ad-hoc manner.

For example, a scheduler design specification may not clarify when and how the design processes queue data for a scheduler session. Does the scheduler session start when the data is first queued or just before the design pops the queue? The description may be at a mathematically level of abstraction rather than a design layer of abstraction. A verification environment's predictor model may need to understand when queue data enters the scheduler session in order to proper model the scheduler's behavior and predict results as well as properly update functional coverage data. Lacking a clear description may not affect the design's functionality but make it very difficult to prove the functionality is working properly within the verification environment. It is best to clarify these details with the designer and then request that they update the missing information in the Hardware Specification. Ideally, action items should track all design specification deficiencies. As a last resort, the verification plan may need to include information that clarifies the design if the designer refuses to update the hardware documentation. It is a bad idea for the verification engineer to investigate the RTL code to determine how a design operates. This may bias the verification engineer's testing strategy and diminishes the redundancy of the interpretation of the specification. Additionally, not properly fixing hardware documentation issues will trickle down and may cause inefficiencies to others involved software, diagnostics, etc.

The luxury of informal commutation is less likely to occur if engineers are working across multiple sites and especially if there are global time zones factors. Therefore, it is crucial that the design documentation and verification planning is up to a higher standard for multiple site projects as well as setting up more reviews.

3.7 Mentoring Verification Teams

The verification team often adds new members at various junctures of the project. During this time, the new verification team members need to get up to speed very quickly in order for the verification effort to benefit from economies of scale. We have found that it is beneficial for a verification project to include an intranet page or at least a document with at least the following information:

(1) How to setup the EDA tools and IT accounts
(2) URLs to all the internal/external design specifications and verification plan
(3) Setup and run a test or a regression in a sandbox instructions
(4) Code check-in procedure
(5) How to collect functional coverage and code coverage and where the resulting results are stored
(6) URL to project coding guidelines

By having this information immediately available it helps augment bring new folks up to speed and reduces the amount of time needed for other verification teams members to answer basic project procedure and methodology questions. This allows the verification team to have more time to concentrate on verifying the design. If any of the information in the intranet page is not current or needs clarification, we have the new team members update the information.

4 Instrumenting Functional Coverage

Verification engineers implement functional coverage points manually inside the verification environment. The functional coverage points should be implementation only in the passive code; in other words the "monitors". Both the unit and system level testbenches always utilize the

monitors but not necessarily the generators, drivers, or bfms. The figure below depicts a situation where a SPI bus interface is driven and monitored in the unit level testbench but the system-level testbench only monitors the SPI bus interface. In both the system and unit level testbenches, functional coverage needs to be gathered.

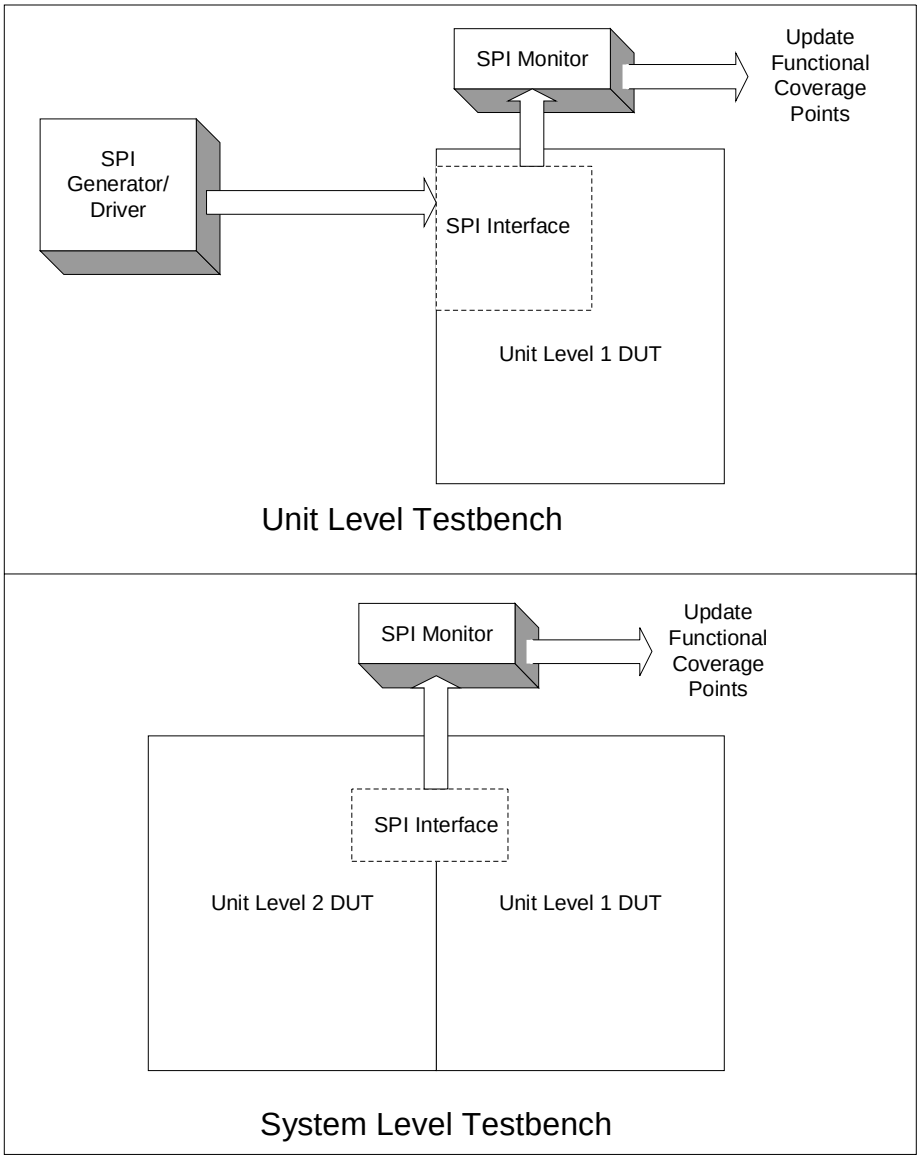


Figure 3 Unit and System Level Testbenches

Although it takes time to instrument functional coverage points, most properly developed random verification environment already have most or all the variables available to code up the necessary functional coverage points. We found coding functional coverage points is a relatively easy task in the entire scheme. The verification team should implement the functional coverage code while they are implementing the verification environment rather than trying to retrofit the coverage points in after building the entire verification environment. As discussed latter in this paper functional coverage implementation may be tracked by tools.

4.1 Accessing functional coverage points from multiple components

Often it is necessary to perform cross coverage with functional coverage points that are included in one ore more verification components. In order to accomplish this we setup a coverage class in the top-level verification component that has a pointer that has access to the entire verification environment. Within the coverage class, we developed coverage groups that had access to multiple verification components. Care must be taken when crossing coverage points with different sampling events. At times, we generated new coverage groups that sampled data from multiple coverage groups within a certain valid temporal window.

4.2 When to update functional coverage

False positive testing is a huge concern for a verification effort. Are all the tests that say PASS really testing anything? Just because a functional coverage point indicates that a certain feature has been exercised it does not mean that it was properly checked. Even if you implement checkers how are you assured they are actually being executed. Therefore, it is paramount that the verification team takes all possible steps to limit the likelihood of false positive testing.

When possible, it is best to update functional coverage points based on a positive response from a checker that verifies the expected behavior against the actual design behavior. This is one way to raise confidence against false positive testing. The example below demonstrates how the dllp functional coverage point is not updated until the checker verified the packet type.

```
if (pkt.type !=  DLLP_REPLAY) {
        error("Design sending unexpected packet type");
}
else {
        update_dllp_functional_coverage(pkt);
}
```

Figure 4 Functional Coverage Update Example

In addition, by updating configuration settings only when a monitor observes activity that affect of that configuration setting is another way to reduce false positive testing concerns. For example, if a bus can operate in either 16-bit or 32-bit mode, then do not update the functional coverage point at the beginning of simulation when setting up the design configuration. Instead, update the functional coverage point only after waiting for a transaction on the bus that verifies the 16-bit or 32-bit configuration mode is operating as expected.

4.3 Performance considerations with functional coverage

Functional coverage points should be updated at the highest level of abstraction that is possible and only updated only when changes occur. In other words, avoid updating functional coverage points on a cycle-by-cycle basis or when unnecessary. Attempt to update functional coverage points only when the appropriate meaningful data is available. Following these rules may help avoid running into simulation performance issues.

Specman e, Vera, and System Verilog contain constructs that allow users to turn on and shut off functional coverage. In some test environments, we added controls to allow our regressions to shut off functional coverage. Although we never ran any formal benchmark testing to determine what the performance hit was for running with functional coverage, we found that running simulations with functional coverage turned on appeared to have a minimal affect on performance. It seems that running simulations with waves and noisy file logging has a more detrimental affect to simulation performance.

4.4 Asserting Errors

Languages such as Vera and Specman e offer verification constructs that allow users to code up functional coverage points that will assert errors when certain conditions occur. We feel that it is best to separate functional checkers and functional coverage points. This allows for a modular and cleaner implementation. Even more importantly, functional coverage may be turned off which may result in false positive testing.

4.5 Using enumerators helps

Item functional coverage points are variables that may be of type integer, bits, or enumerators. We found that if a coverage-point is not used in the context of ranges then it wise to use enumerator types. Enumerator types help clarify the resulting functional coverage data in the functional coverage database. For example, if you have a functional coverage point for a configuration field for 16-bit or 32-bit mode it is best to implement an enumerator type with values MODE_16_BIT and MODE_32_BIT rather than using a bit type. When looking at the functional coverage data it is much more intuitive to read MODE_16_BIT and MODE_32_BIT compared to a 0 and 1. Additionally, the code implementation tends to be easier to read/maintain and the compiler performs type checking on the enumerators.

5 Testing Strategies for Functional Coverage

In a random verification environment, it is optimal to develop as few tests as possible and make the tests very random in order to hit as many of the functional coverage points as possible. When tests have a higher degree of randomness more of the design's verification space is exercised. We advocate that it is ideal to keep a test's simulation time under 20 minutes if possible and rerun the tests multiple times with random seeds. Long tests are often more difficult to debug and wave files may grow extremely large.

Nearly always it is impractical, too difficult time consuming, ands/or inappropriate to have a single unconstraint random test case that verifies all the design features. For example, it may be inappropriate for a rate test to send random error packets. Therefore, the verification team needs to develop a random rate test that is similar to the unconstrained random tests except it does do not send error packet scenarios. A properly developed random verification environment will provide verification engineers extensible constraints so that it is very simple for a test to control what random scenarios the generator produces – ultimately a test consists of a set of knobs that control the entire testbench.

Initially, we usually develop several highly constraint baby tests that verify certain features and then slowly add the features to the unconstrained random test. We find debugging is easiest by limiting the amount of stress and features at first. Finally, we add the features into the unconstraint random test and retire the baby test after we verify that the baby tests passes against a hundred seeds.

5.1 Are tests random enough?

It is possible to measure the randomness of a test using a *test ranking*. *Test ranking* results helped us uncovered issue where a verification team was developing tests with low randomness. We experienced situations where a verification team developed many tests that were highly constrained in order to hit most of the functional coverage points in the functional coverage plans and they considering their testing complete. By using test ranking, we uncovered that most of the tests were hitting only a few coverage points. Therefore, we directed the team to develop random tests with few or no constraints closed. After enhancing the tests randomness, the team started uncovering hard-to-find corner-case design bugs and the test ranking numbers increased.

5.2 Developing directed tests

Occasionally when trying to close functional coverage hitting certain random conditions is very difficult even after testing with many random seeds. It may be necessary to tweak constraints or the verification team, as a last resort, may need to develop a test case to verify the hard to hit scenario. The new test case should be as random as possible except for the constraint(s) that controls hitting the scenario in question. Finally, the new test is added to the regression but by no means should the original random test be removed. Remember the idea for random testing is to maximize exercising and verifying the design's verification space.

5.3 Developing coverage reactive tests to hit coverage holes

Verification languages such as Vera, System Verilog, and Specman e have constructs that allow users to examine the functional coverage database dynamically during simulation. This information may be feedback and control the stimulus drivers and/or ending the test. However, these approaches may work for a trivial test case but is not scalable to operate in a large-scale verification effort. Additionally, it may biases the generation engine to a restricted area within the verification space if the feedback controls stimulus. Controlling test ending based on functional coverage goals may be a bad idea because functional coverage may be turned off. All verification efforts we have been involved in use random shotgun stimulus with manual functional coverage feedback. Reactive feedback testing from a regression rather than simulation level and requires no user interaction is currently being researched.

6 Tracking Functional Coverage Progress

Tracking functional coverage closure progress is often a major challenge. During the early phases of development, the verification team is usually developing verification infrastructure and in the latter stages, they are busy try to close functional coverage. It is difficult to measure verification infrastructure progress and it is time-consuming error prone process to measure functional coverage data results.

In the past, we have develop tools or used existing EDA tools that read in coverage plans and tie them directly into the regressions results. The tools graphically display where coverage holes existed. The tools also calculated a "doneness" score for the functional coverage plan's coverage points. This gave management visibility on where the verification team's progress was towards closing functional coverage for all the testbenches. Additionally, the tools gave us some insight in the infrastructure development side by tallying up which functional coverage points where not implemented.

Without a functional coverage tool, the verification engineer would need to traverse a raw functional coverage report manually. The raw functional coverage report presents all functional coverage points; many which may not be pertinent because they may belong to other areas of the verification environment or some uninteresting verification IP functional coverage points. Furthermore, the verification engineer would need to calculate the functional coverage grade manually using only the appropriate functional coverage points. This process is tedious, error prone, and not scalable.
We developed a tool at Paradigm Works called SystemVerilog FrameWorks™ Functional Coverage Tool. The tool reads in a functional coverage plan from Microsoft Word, Adobe Frame, Excel or Raw Text using a generic XML interface. Then the tool extracts information for all the coverage points from a SystemVerilog Functional Coverage database and then generates interactive HTML Functional Coverage HTML Pages for analysis as well as producing PDF report files.

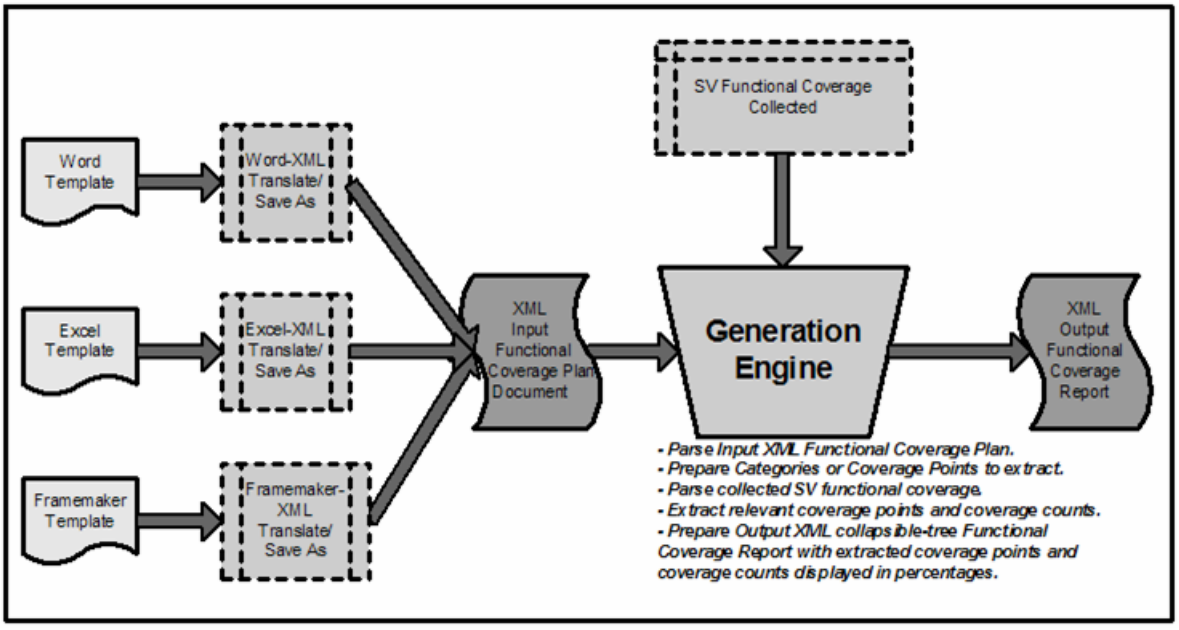


Figure 5 SystemVerilog FrameWorks™ Functional Coverage Tool Flow

Shown below (Figure 6) is an example of a Word formatted functional coverage plan for a transmitter section is. The user simply enters in functional coverage points and optional kind, weigh and goal functional coverage attribute information. The ID column is a mapping key to the functional coverage database. Note that the tool checks the integrity of the implementation of the functional coverage point. For example, if the functional coverage point feature_4 is the same as

the functional coverage point feature_3 the tool displays a warning. The tool also performs sanity checks for kinds, weights, and goals that do not match the implementation.

Name	Kind	Description	Values	Weight	Goal	Id
feature_1	**ITEM**	**This coverage point shows which input address ranges have been exercised.**	**SHORT, MED, LONG**	**1**	**100**	**transaction.range_cp**
feature_2	**ITEM**	**This coverage point shows which input address ranges have been exercised.**	**HIGH, LOW**	**1**	**100**	**transaction.data_cp**
feature_3	**CROSS**	**This coverage point shows which input address ranges have been exercised.**	**HIGH/SHORT, HIGH/MED, HIGH/LONG, LOW/SHORT, LOW/MED, LOW/LONG**	**1**	**100**	**transaction.cross_ad_rg**
feature_4	**CROSS**	This is the same as above! Usually, a coverage plan should not include the same coverage point. The svfFC Tool will display a WARNING when users define the same coverage point!	**HIGH/SHORT, HIGH/MED, HIGH/LONG, LOW/SHORT, LOW/MED, LOW/LONG**	**1**	**100**	**transaction.cross_ad_rg**

Figure 6 SystemVerilog FrameWorks™ Functional Coverage Tool Transmitter Plan Example

The format of the generated HTML report matches that of the Functional Coverage Plan. The HTML displays two panels; an interactive tree control panel and a display panel – see Figure 7. The tree control panel allows the user to navigate the Functional Coverage Plan's categories (sections) by expanding or collapsing categories or by selecting a summary page. The tree control and summary are pessimistically color coded so users can quickly determine where functional coverage holes exist as shown in Table 6.

Color	Description
GRAY	Coverage points are GRAY if any of the coverage points are **NOT** implemented regardless of any of the implemented coverage points grades.
RED	If all the coverage points are implemented and they all have a grade is zero then the section is colored RED.
YELLOW	If all the coverage points are all implemented and the grades are inclusively between 1 and 99 then for all the coverage points then the section is colored YELLOW.
GREEN	If all the coverage points are implemented and the grades

	are all 100 then the section is colored GREEN.

Table 6 SystemVerilog FrameWorks™ Functional Coverage Tool Color Chart

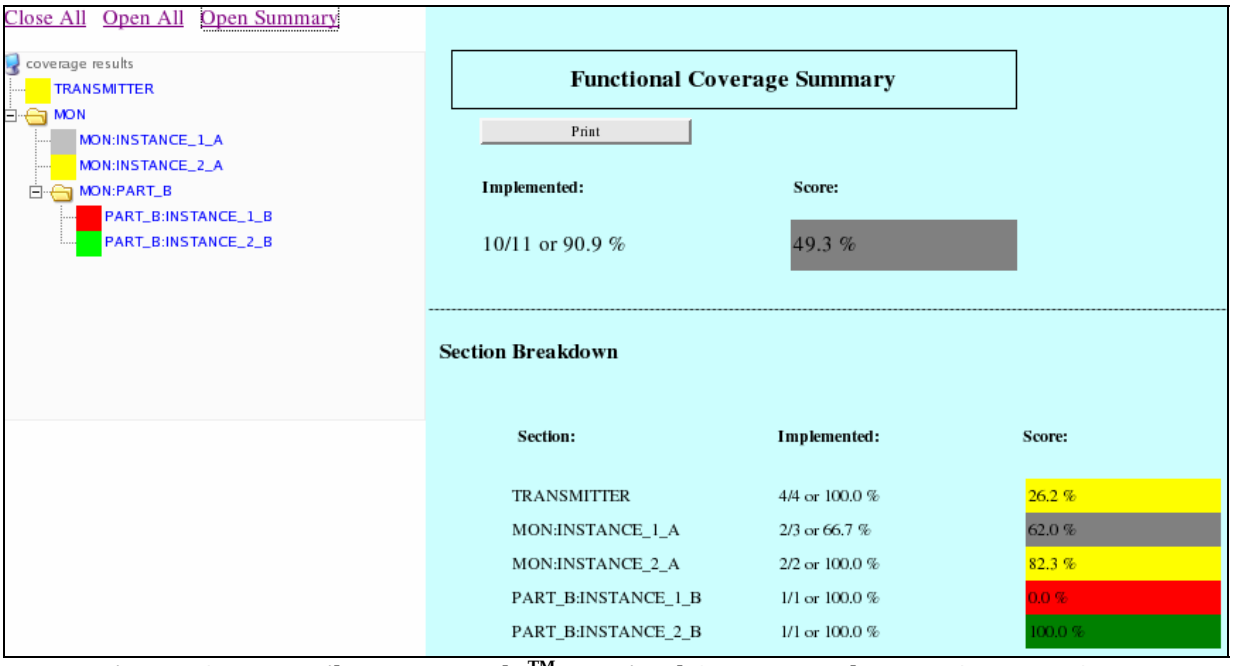


Figure 7 SystemVerilog FrameWorks™ Functional Coverage Tool HTML Summary Output

When the user selects a section on the tree, all the data for those functional coverage points are shown on the display panel – see Figure 8 to see an example of the transmitter coverage plan's results. This output clearly identifies areas that contain holes for the user, which in return will expedite closing functional coverage. The PDF Functional Coverage reports are valuable for the management team to have visibility of the verification team's progress. Additional, the tool generates reports for SV Coverage point code implementation.

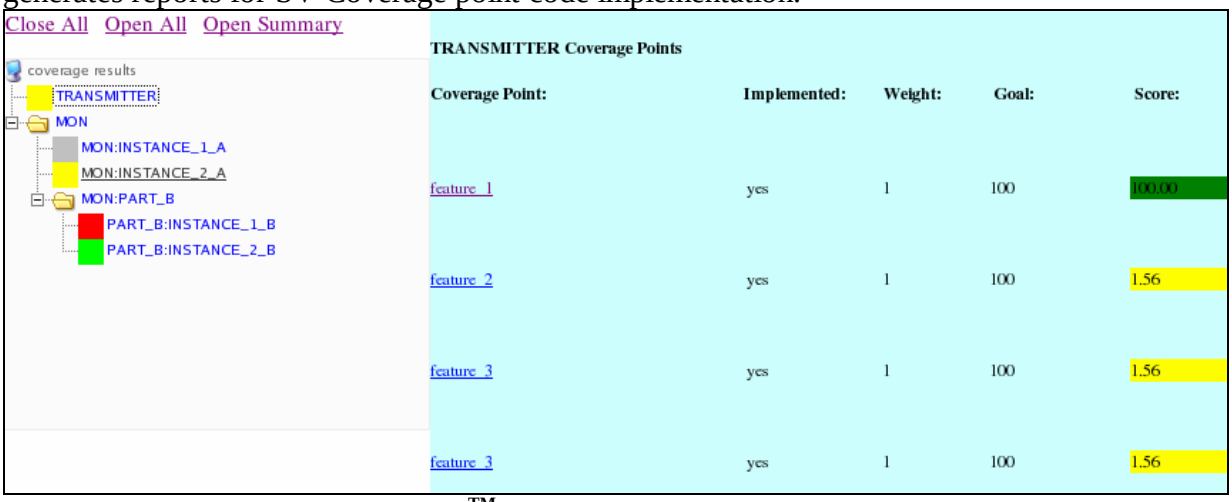
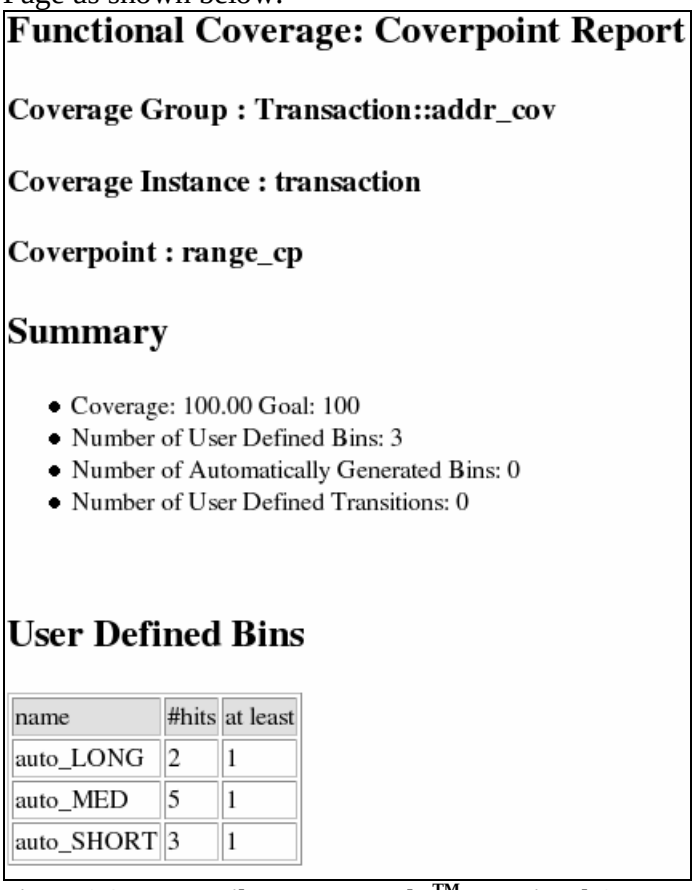


Figure 8 SystemVerilog FrameWorks™ Functional Coverage Tool HTML Transmitter Section Output

Additionally, Synopsys Functional Coverage Point statistics are displayed when the user clicks on the coverage point URL on the SystemVerilog FrameWorks™ Functional Coverage Tool HTML Page as shown below.



Functional Coverage: Coverpoint Report

Coverage Group : Transaction::addr_cov

Coverage Instance : transaction

Coverpoint : range_cp

Summary

- Coverage: 100.00 Goal: 100
- Number of User Defined Bins: 3
- Number of Automatically Generated Bins: 0
- Number of User Defined Transitions: 0

User Defined Bins

name	#hits	at least
auto_LONG	2	1
auto_MED	5	1
auto_SHORT	3	1

Figure 9 SystemVerilog FrameWorks™ Functional Coverage Tool Links to Synopsys Coverage Info

7 Conclusion

Functional coverage metrics provide valuable insight into which design features that the random verification environment tested. Functional coverage is at the design specification level of abstraction; this is appropriate for functional verification. Without using functional coverage, the verification team needs to develop many highly constrained tests to have some level of assurance that design features are being exercised. This is contrary to the original ideology for developing the random verification environment.

Although it takes time to instrument functional coverage points, most properly developed random verification environment already have most or all the variables available to code up the necessary functional coverage points. We found coding functional coverage points is a relatively easy task. Additionally, we gain some confidence against false positive testing by updating functional coverage metrics after associated functional checks.

It is vital to have a verification plan that includes functional coverage plans that identify all functional coverage points and describes a methodology for merging functional coverage data. The functional coverage plans are a large part of the “doneness” criterion for the verification

effort. Appropriate design, architecture, and project team members should sign-off these plans. The plans must remain as "living" documents and be updated when functionality changes. After the team reviews the plans, the verification team should not change the functional coverage plans unless appropriate approval is granted.

The verification team needs to track functional coverage by *calculating grades* and *identify functional coverage holes* based on the functional coverage points included in the functional coverage plans. Additionally, functional coverage data from multiple passing regression runs need to be *merged* together. HVL languages (including System Verilog) do not have built-in features that directly help with this task. The verification team needs to merge functional coverage results and parse through functional coverage reports manually to determine where they are in closing functional coverage. This process is tedious, error prone, and not scalable. The Paradigm-Works' SystemVerilog FrameWorks™ FC Tool eliminates all these issues by generating interactive HTML regression reports based on the functional coverage plan in a Vera and SystemVerilog environment.

---

[i] Writing Testbenches: Functional Verification of HDL Models Second Edition by Janick Bergeron

[ii] Verification Methodology Manual for System Verification by Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale