Molding Functional Coverage for Highly Configurable IP

J. Ridgeway[1] K. Chaturvedula[2] K. Dhruv[3]

[1]Avago Technologies, Ltd., 4380 Ziegler Rd., Fort Collins, CO, 80524, jeremy.ridgeway@avagotech.com;
[2]Avago Technologies, Ltd., 1320 Ridder Park Dr., San Jose, CA, 95131, kavitha.chaturvedula@avagotech.com;
[3]Consultant, karishma.dhruv@gmail.com

***Abstract*-It is hard to accurately define a functional coverage model for intellectual property (IP) cores with many top-level static (compile-time) and runtime (modes for simulation) configurations. One approach develops a super-set model and waive, in post-regression analysis, scenarios deemed not applicable. Since, customer requirements specify the IP static configuration, a customer-specific model may be generated from the super-set prior to simulation. In this paper we present a methodology transforming a super-set abstract functional coverage model to a customer-specific implementation. We show how the model can change dramatically between customers. Our approach was successfully employed in an IP project supporting simultaneous customers.**

I. Introduction

It is hard to accurately define and maintain a functional coverage model for intellectual property (IP) cores with many different configuration options. The customer-specific IP static configuration is usually chosen by compiler directives (e.g., `define macro). Within a single static configuration, the IP may further operate in disparate modes, often chosen at hardware reset-time. For constrained random verification, the IP mode of operation is chosen at simulation time zero by randomizing a global configuration object. The IP mode of operation may affect both hardware and test bench components.

For example, consider a PCI-Express based end-point device IP. The Link Training and Status State Machine (LTSSM) in the PCIe logical physical layer controls access to the link for data transmission and reception. Low power is one feature that affects a significant portion of both the hardware and test bench. The low power mode, when supported, is often chosen at hardware reset-time and/or simulation time zero.

The PCIe Generation 3 standard identifies three active state power management (ASPM) options: L0s, L1, and L1PM-substates (there are other PCIe low power, but we will focus only on these) [1]. A PCIe IP may support either no low power or a combination of low power options. When low power is enabled in a specific mode at runtime it has bearing on the testing coverage. In other words, it is not sufficient to ask: "have I covered all LTSSM states in testing?" Instead, the question should be posed *in the context* of the low power capability and mode of operation selection.

Consider when a PCIe IP does not support low power. Then the low power LTSSM states are not applicable and require no functional coverage. The context of "no low power support" dictates that no low power coverage is necessary. Now consider when a PCIe IP supports any combination of low power but these are all disabled. Then an error should be reported if the IP enters any low power LTSSM state. Functional coverage for the context of "low power disabled" mode-of-operation should *not* include the arc from the LTSSM active state, L0, to a low power state, L0s or L1. Doing so skews the overall coverage score.

For example, consider some LTSSM state functional coverage reporting the following transitions:

$$\{ \text{detect} \rightarrow \text{polling}, \text{polling} \rightarrow \text{configuration}, \text{configuration} \rightarrow \text{L0}, \text{L0} \rightarrow \text{Recovery}, \text{L0} \rightarrow \text{L0s}, \text{L0} \rightarrow \text{L1} \}.$$

Suppose, too, that low power is supported (L0s and L1 included) but always disabled at simulation time zero. First, there is no set of tests to fully cover the indicated transition space. The achievable group coverage score in this mode is 4/6 transitions, or 66.67%, dragging down the overall coverage score (false-negative coverage).[1] Second, if a transition to L0s or L1 does occur, and the functional coverage reporting is not excluded, then the achieved coverage score is 5/6 transitions, or 83.33%, pulling up the overall achievable coverage score (false-positive coverage). Instead, when the "low power disabled" context is applied, then low power transitions must be excluded.

[1] We acknowledge that it may be possible to properly weight or waive transitions for this mode-of-operation, but we assert that this hand-tuning is not ideal.

For low power, instead of reporting functional coverage for all low power LTSSM states, and weighting or waiving later, we wish to report that all *applicable* low power LTSSM states have been covered. In other words, functional coverage must consider:

- Overall low power support,
- L0s support,
- L1 support,
- L1PM-substates support,
- Overall low power enabled,
- L0s enabled,
- L1 enabled, and
- L1PM-substates enabled.

The PCIe standard reflects this dichotomy with capability registers (support) and control registers (enable/disable). Considering just LTSSM state coverage, writing and maintaining individual functional cover models for each customer could become a time consuming task.

Moving beyond LTSSM state coverage, low power support may have bearing on other components of the verification environment, too. Therefore, low power coverage should not simply be modeled as a single cover group. Instead, low power capability and control may be incorporated into all applicable cover groups in the functional coverage model by crossing the bins in an applicable cover group with the low power bins.

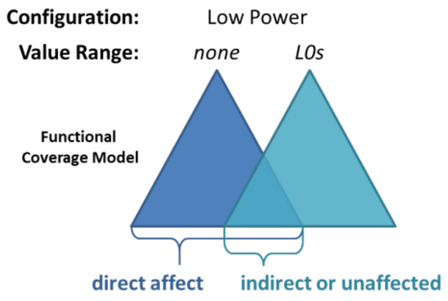


Fig. 1: Low Power root-level coverage points may have bearing on other cover groups dependent on the selection. A root-level point bin value may have a direct effect on cover groups (i.e. is crossed), indirect (i.e. conditionally crossed), or no affect.

In Fig. 1, we have identified Low Power as a root-level coverage point. That is, the cover groups beneath the root are all covered "in the context of" the root-level point. This correlation may be direct, indirect, or no correlation (unaffected). A direct correlation applies when the points in a cover group are crossed with the root-level coverage point. In Fig. 1, none and L0s are two bins for the Low Power cover point. A directly affected cover group crosses its points with Low Power bin none and Low Power bin L0s, separately. A directly affected cover group would abstractly exist in both solid sections of the triangles in Fig. 1, indicating its crosses are applicable to both mode values separately. An indirect correlation implies that the cover group is conditionally crossed or crossed with a with a subset of the root-level bin values. For example, a cover group that monitors *any* lower power mode active has an indirect correlation. Finally, unaffected cover groups are outside of the scope of the root-level coverage point. These cover groups are not crossed with the root level point (or a cross with all values in a single bin).

Now suppose that the IP design supports all four potential low power options: none, L0s, L1, and L1PM-substates. If some customer selects only low power disabled and L0s, then the functional coverage model may undergo a transformation. In Fig. 2-A, the Low Power configuration option has the full range of values. The customer supported model, in Fig. 2-B, indicates only off (i.e. none or none enabled) and L0s is available.

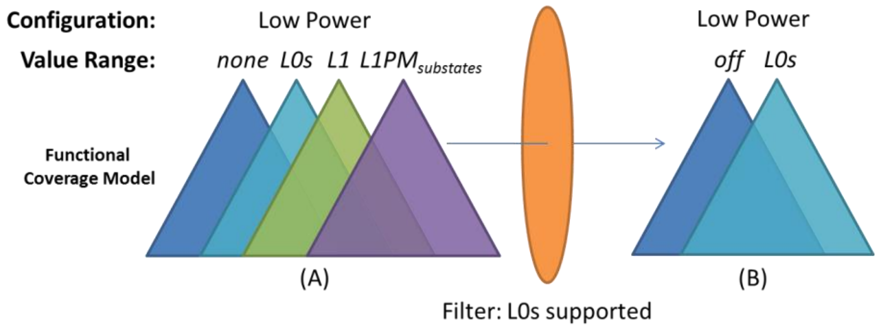


Fig. 2: Static configuration options (A) become runtime configurations (B) after passing through the customer requirements filter.

From a static configuration perspective, any functional coverage point or cross that includes low power options L1 (green triangle) and L1PM-substates (purple triangle) is no longer applicable. However, in a constrained random verification environment separation of the valid and invalid cover points or crosses may be difficult. As such, *all* points and crosses related to IP customer-unsupported low power options would need to be waived. Or, as in our approach, the functional coverage model itself may be transformed, referring again to Fig. 2, such that L1 and L1PM-substates cover point bin values no longer exist.

From a mode of operation perspective, any point or cross covered in simulation when low power was disabled (at time simulation time zero) makes no coverage statement *of that same point or cross* when low power L0s is enabled. In a constrained random verification environment, it may not be clear how to differentiate such coverage. Or, as in our approach, the functional coverage model itself may be transformed, referring again to Fig. 2, such that applicable cover point bins are crossed with the applicable low power mode of operation point bin values.

The remainder of this paper presents our approach to molding a hierarchical functional coverage model for highly configurable IP via script automation. In section II, we compare where our work fits with related functional coverage work. In section III we discuss the cover model architecture while in section IV we dig into a few important implementation details and cover model optimizations. We discuss our current experience and some limitations with the work in section V. Finally, we conclude in section VI.

II. Related Work

It is well known that coverage-driven constrained random verification is effective in measuring progress and providing confidence in both the hardware and verification quality [2], [3], [4]. Motivations behind the functional coverage model and guidance on its development and reporting structure are described in [5] and [3]. However, few guidelines currently exist that target highly configurable IP designs with potentially disjoint functional coverage spaces. Furthermore, while all major simulator vendors provide verification planning tools, [6], [7], [8], these are insufficient for fine grain cover group planning (point bins, crosses) and hierarchical model reuse.

Current runtime approaches to coverage-driven verification look to balance testing stimulus generation based on cover groups and parameter-domains [9]. Alternatively, they may combine the coverage model and constraints to build a coverage driven random distribution [10]. Formal analysis also work with existing coverage to classify [11], analyze [12], [13], and/or iteratively guide test stimulus [14], [15], [16]. With the exception of [10], these techniques focus on post-simulation analysis to produce more concise coverage reports and/or to guide test stimulus in the next iteration. They all assume a suitable functional coverage model is readily available.

The approach we present here is targeted specifically to planning functional coverage in a hierarchical manner (enabling reuse) and automatically including or excluding entire branches of the model depending on the capabilities of the design. As we focus the functional coverage model itself, it is compatible with runtime approaches to achieve desired coverage driven verification goals.

III. Hierarchical Abstract Functional Coverage Model

Design IP configuration variables root our abstract functional coverage model. In Fig. 1 and Fig. 2, specific bin values of the configuration cover point, Low Power, root each model. The generalized form, as depicted in Fig. 3, is a tree composed of **cover blocks**. The *root block* of the tree is the top-most cover block, while the *leaf blocks* have no descendants. Functional coverage is organized in cover blocks as a related collection of cover groups and child cover blocks. There is no limit to the depth of the tree and cover blocks may have multiple parents. The implication

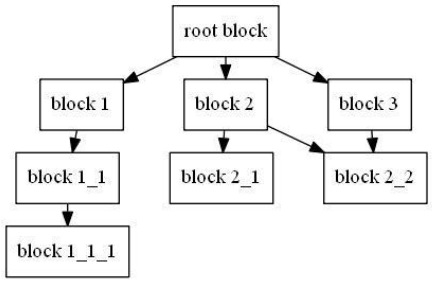


Fig. 3: Hierarchical functional coverage model represented as an abstract tree. Arrows point from the parent block(s) to its(their) descendants.

with block 2_2 is that its cover groups is related to both parents, block 2 and block 3. A **block model** is the sub-tree starting from some cover block to all child leaf blocks. The block model defines a *cone of functionality* to cover in simulation. When the top-most block in a block model indicates complete coverage then the entire cone has been one hundred percent covered. The **cover model**, therefore, is the block model starting at the root block. Verification concludes when the cover model is completely covered.

The abstract hierarchical functional cover model is extensible and reusable. The tree can be extended at the root, the leaf, or in-between. Blocks can be added to leaves to further refine the functionality cone. Within the tree, new parent-child relationships can be established by inserting blocks. Finally, a new root may be established. For example, a cover model for an IP may be imported into a cover model for a subsystem instantiating that IP. The IP-level root block simply becomes a subsystem-level cover block in a tree with a new root.

Similarly, the abstract functional cover model may be transformed. Given an IP-level functional coverage model, such as Fig. 2, we may apply values to configuration variables that result in a model subset. The resultant subset is also a cover model, but tailored to a specific configuration of the IP.

A. Cover Variables

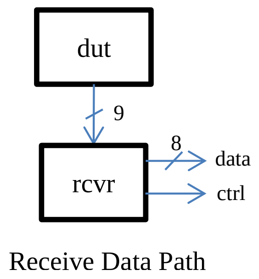
The basic element of a SystemVerilog functional coverage group is the cover point and its value bins. The cover point is bound to a test bench variable or device under test (DUT) signal as a *point of observation*; a monitor. The set of bins in the cover point define the values that must be observed in simulation on the bound variable or signal during the course of verification. Cover point bin values are usually instantaneous (e.g., 32’h1 or [2’b00:2’b11]), but temporal values are supported and indicated with an arrow (e.g., 2’h0 → 2’h3). With temporal values, the bin is covered when the sequence of values are observed in-order during simulation.

We abstract the cover point as a **cover variable**. Every SystemVerilog cover point becomes a type of cover variable in our model. Similar to a SystemVerilog cover point, the cover variable has some symbol name and a range of expected values. However, as depicted in TABLE 1, cover variables offer some flexibility. Variables may be defined in order to classify specific values. For example, ltssm_detect in TABLE 1 defines LTSSM state enumeration values for the detect states. This variable is not bound to any class member or DUT signal. Instead, the cover variable may be used to build other cover variables and again directly within a cover group to for a specific kind of cross. The ltssm_state cover variable, in TABLE 1, incorporates all the ltssm_detect variable values (the dollar-sign indicates another variable should be used with its values substituted). In this manner we achieve two goals: (1) provide a focus on planning instead of implementation (yet) and (2) encourage cover variable reuse. The cover variable is defined separate from any cover group. Then, when a cover variable is *instantiated* in a cover group it becomes a SystemVerilog cover point.

Two Low Power variables are defined in TABLE 1, a configuration-type cover variable, and a mode-type variable. The configuration variable defines the full range of values the IP design can support. Configuration variables, are static (compile-time, `define) options. Just as the SystemVerilog pre-processor eliminates `define macros (by macro expansion), the configuration variables of the abstract functional coverage model are eliminated during script pre-processing. The result is a collection of mode variables with customer-specific values. As mode variables are instantiated in cover groups, they automatically inherit the bound signal.

TABLE 1
PHYSICAL LOGIC LAYER COVER VARIABLES

Name	**Range**	**Signal**	**Description**
C_lowpower	off, L0s_en, L1_en, L1PMss_en		Low Power Configuration
M_lowpower	$C_lowpower	CFG::LP	Low Power Mode of Operation. Bound to global static variable.
ltssm_detect	detect_quiet, detect_active		Enumeration of detect states
…	…		*other LTSSM state enumerations*
ltssm_L0s_RX	L0s_rx_entry, L0s_rx_idle, L0s_rx_FTS		Enumeration of receiver L0s states
ltssm_state	$ltssm_detect, …, L0, $ltssm_L0s, …	tb.ltssm_o	Enumeration of all LTSSM states possible. Bound to DUT output.
ltssm_l0s_rx_trans	L0 → L0s_rx_entry, L0s_rx_FTS → L0	tb.ltssm_o	Single low power transitions.



```
class MonitorA;
  logic [7:0] data;
  logic ctrl;
  covergroup rx_dp_cg;
    coverpoint data;
    coverpoint ctrl;
  endgroup
endclass
```

```
class MonitorB;
  logic [7:0] data;
  logic ctrl;
  covergroup rx_dp_cg;
    coverpoint data;
    coverpoint ctrl;
    cross data, ctrl;
  endgroup
endclass
```

Fig. 4: Functional coverage on receive data path. Monitor_a provides no context; there is no correlation between the signals. Monitor_b is over contextualized; the correlation includes invalid bins.

B. *Cover Groups*

The cover group collects correlated cover points while cover point bin values are crossed to provide meaning to the coverage. It is not enough to simply instantiate cover points within a cover group. Instead, context to the coverage is achieved by crossing relevant cover points values. For example, consider a data path with both control and data characters, such as an 8b/10b encoded data stream, Fig. 4. Received control characters are indicated when the ctrl signal is asserted, otherwise the bus indicates a data character. The cover points in MonitorA have no correlation; there is no way to discern control character coverage even if both cover points are completely covered. The opposite is MonitorB which simply crosses everything. Even if both cover points are completely covered there is no way for complete coverage on the cross because many invalid control characters are expected. Instead, the cover group should be refined to exactly the coverage required.

TABLE 2
COVER VARIABLE DEFINITION FOR CONTROL CHARACTER MONITOR

Name	Range	Signal	Description
Data	[8'h00:8'hff]	data	Decoded data bus from DUT
Control	0, 1	ctrl	Control character indication
COM	8'hBC		Comma control character, K28.5
STP	8'hFB		Start transport packet, K27.7
SDP	8'h5C		Start data-link packet, K28.2
END	8'hFD		End of Packet, K29.7
ControlChars	$COM, $STP, $SDP, $END		Decoded control characters

In our functional coverage architecture, each point is first defined as a cover variable. In TABLE 2, the Data and Control variables are, by default, bound to the class members data and ctrl, respectively. We have further defined some variable aliases for describing specific control characters, COM, STP, SDP, and END. These are not bound to any signal or class member as they are only used within coverage model.

The cover group is defined as a table, refer to TABLE 3, and in a similar manner as described by [5]. Each cover group has a name and declares the cover variables to instantiate (rows 1-2). Each of the remaining rows of the cover group (rows 3-8) defines a single cross (or group of crosses) of cover point bin values. Blank cells on a row indicate no correlation (i.e., not crossed); the wild card, '*', indicates complete correlation (i.e., cross all values); otherwise, the cross correlation is described. For example, MonitorA is sufficiently described by rows 3 and 4. There is no correlation between the cover variables Data and Control and, thus, the cover group contains only cover points and no crosses. MonitorB is sufficiently described by row 5; two cover points and a complete cross, including invalid control characters. A more refined cover group for the receive data path in Fig. 4 covering only the interesting control characters in TABLE 2 is described by row 6. The equivalent Boolean formula for row 6 is

$$\textit{any\_ctrl\_cross} \leftrightarrow (\text{Data} == 8\text{'hBC} \,||\, \text{Data} == 8\text{'FB} \,||\, \text{Data} == 8\text{'h5C} \,||\, \text{Data} == 8\text{'hFD}) \;\&\&\; \text{ctrl} == 1\text{'b1}. \quad (1)$$

Now coverage is limited only to interesting control characters but they are not all valid in all LTSSM states. Therefore, rows 7-8 provide additional context to the control character coverage. Packets may only be initiated in L0 state. Cover variable ltssm_state, from TABLE 1, is also instantiated in the cover group and crossed in row 7 to indicate packet delimiter coverage, and again in row 8 to indicate low power exit control character coverage. We have successfully reused the ltssm_state cover variable and bin values in the cover group.

TABLE 3
COVER GROUP EXAMPLE DEFINITION FOR CONTROL CHARACTER MONITORS

1	**Covergroup Name**	rx_datapath_cg			
2	**Cover Points**	**Data**	**Control**	**ltssm_state**	***Comment***
3	*data coverpoint*	*			Cover points equivalent to
4	*ctrl coverpoint*		*		Fig. 4, MonitorA
5	*data_ctrl_cross*	*	*		Coverage as Fig. 4, MonitorB
6	*any_ctrl_cross*	$ControlChars	1		Refined coverage monitor
7	*pkt_delim_cross*	$STP, $SDP, $END	1	L0	Cover packet delimiters
8	*L0s_wake_rx_cross*	$COM	1	L0s_rx_FTS	Cover FTS receive

C. *Config and Mode Variables*

The *L0s_wake_rx_cross* at row 8, TABLE 3, is sufficient for the internal PCIe IP design but it now must be waived for a customer configuration that does not support L0s low power mode. The Boolean formula for row 8 is

$$\textit{L0s\_wake\_rx\_cross} \leftrightarrow \text{Data == 5'hBC \&\& ctrl == 1'b1 \&\& ltssm\_state == L0s\_rx\_FTS}. \qquad (2)$$

The LTSSM will never enter L0s_rx_FTS when L0s is unsupported. In fact, this is a common occurrence because when a PCIe device is "operating with separate reference clocks with independent Spread Spectrum Clocking (SSC), L0s is not supported" [1]. In these configurations, the *L0s_wake_rx_cross* cannot be covered. We handle this situation in our coverage flow by introducing the mode and configuration variables, as in TABLE 4.

TABLE 4
REFINED COVER GROUP FOR VALID CONTROL CHARACTER MONITOR, READY FOR TRANSFORMATION

1	**Covergroup Name**	rx_datapath_cg				
2	**Cover Points**	**Data**	**Control**	**ltssm_state**	**M_lowpower**	**C_lowpower**
3	*pkt_delim_cross*	$STP, $SDP, $END	1	L0		
4	*L0s_wake_rx_cross*	$COM	1	L0s_rx_FTS	L0s_en	L0s_en

A **config variable** is used similar to a band-pass filter in our coverage flow. Effectively, the config variable validates the cross scenario. That is: "this cross scenario is valid for the customer configuration when the config variable range includes the specified value or values." The whole functional coverage model may be contracted by restricting configuration variable values.

Like SystemVerilog compiler directives, config variables are used only by our coverage script; they do not propagate to generated SystemVerilog cover groups. For example, the C_lowpower config variable is defined in TABLE 1 with the full range of supported values for the internal IP. Assuming the C_lowpower config variable range is set, at script-time, to reflect the customer configuration, then it may be used as a filter. As such, our coverage script allows modification of config variables on the command-line. For some customer that does not support L0s low power, the C_lowpower config variable would *not* include the value L0s_en. Then, our coverage script will not generate the *L0s_wake_rx_cross* in the resultant SystemVerilog cover group. The cover group in Listing 1 reflects TABLE 4 when L0s low power is not supported. Notice that the mode variable M_lowpower is not part of the cover group. The cell entry for *pkt_delim_cross* is empty to indicate there is no correlation with M_lowpower, and thus is excluded from the cover group. Also note that `data_0`, in line 2, defines three bins, one for each value listed. Therefore, `pkt_delim_cross` defines three cross bins, one for each data_0 bin.

```
1 covergroup rx_datapath_cg;
2   coverpoint data { bins data_0[] = { 8'hFB, 8'h5C, 8'hFD }; }
3   coverpoint control { bins control_0 = { 1; }; }
4   coverpoint ltssm_state { bins ltssm_state_0 = { L0 }; } }
5   c_0: cross data, control, ltssm_state {
6     bins pkt_delim_cross = binsof(data.data_0) && binsof(control.control_0) &&
7                            binsof(ltssm_state.ltssm_state_0);
8   }
9 endgroup
```

Listing 1: Generated SystemVerilog cover group from TABLE 4 when L0s low power is not supported.

A **mode variable** is employed at simulation time to provide additional context to the cross. The mode variable expands the overall coverage space for context. For example, assume the customer configuration *does* support L0s low power. In this case, C_lowpower includes the value L0s_en in its range and *L0s_wake_rx_cross* is generated in the resultant cover group. The generated cover group is shown in Listing 2. The data and ltssm_state cover points now have additional bins to support `L0s_wake_rx_cross`.

```
covergroup rx_datapath_cg;
  coverpoint data { bins data_0[] = { 8'hFB, 8'h5C, 8'hFD };
                    bins data_1   = { 5'hBC }; }
  coverpoint control { bins control_0 = { 1 }; }
  coverpoint ltssm_state { bins ltssm_state_0 = { L0 }; }
                           bins ltssm_state_1 = { L0s_rx_FTS }; }
  M_lowpower: coverpoint CFG::LP { bins M_lowpower_0 = { L0s_en }; }
  c_0: cross data, control, ltssm_state {
    bins pkt_delim_cross = binsof(data.data_0) && binsof(control.control_0) &&
                           binsof(ltssm_state.ltssm_state_0);
  }
  c_1: cross data, control, ltssm_state, M_lowpower {
    bins l0s_wake_rx_cross = binsof(data.data_1) && binsof(control.control_0) &&
                             binsof(ltssm_state.ltssm_state_1) &&
                             binsof(M_lowpower.M_lowpower_0);
  }
endgroup
```

Listing 2: Generated SystemVerilog cover group from Table 4 when L0s and off low power modes are supported.

Config and mode variables are defined in a cover block and may be propagated down the tree. Referring to Fig. 3, config and mode variables defined in block 1 affect cover groups in block 1, block 1_1, and block 1_1_1. Config and mode variables defined in the root block affect all cover groups in the model. TABLE 4 explicitly instantiates the M_lowpower mode variable. However, had it not done so the coverage script would automatically and in the most expressive manner (i.e., cross with each value in the mode variable). The implication is that when a mode variable is not instantiated in a cover group then the group should be covered in all possible modes individually. Similarly, when a config variable is not instantiated the implication is that the cover group is valid in all possible configurations. The verification engineer instantiates each explicitly to indicate when the cover group and/or crosses are applicable.

IV. IMPLEMENTATION

We implemented an object-oriented script in Perl5 [17] to process and transform the internal IP design functional coverage model into customer-specific SystemVerilog functional coverage (the orange filter in Fig. 2). Each block in the hierarchical model, Fig. 3, was physically a file system directory. Child blocks, then, were sub-directories therein. Multiple parent relationships were established with symbolic links.

Every block directory contained a Microsoft Excel [18] spread sheet workbook, Cover.xlsx, hereafter called *coversheet*. Coverage variables were defined in worksheets within, one tab for each type of variable: config, mode, and cover. The variable definition style was as indicated in TABLE 1 and TABLE 2, allowing multiple variables defined on one worksheet. Cover groups were also defined in worksheets within the coversheet, in the group tab, and in the style indicated from TABLE 3 and TABLE 4, in the manner described in [5]. Multiple cover groups may be defined on one or multiple worksheets (include one worksheet tab in another).

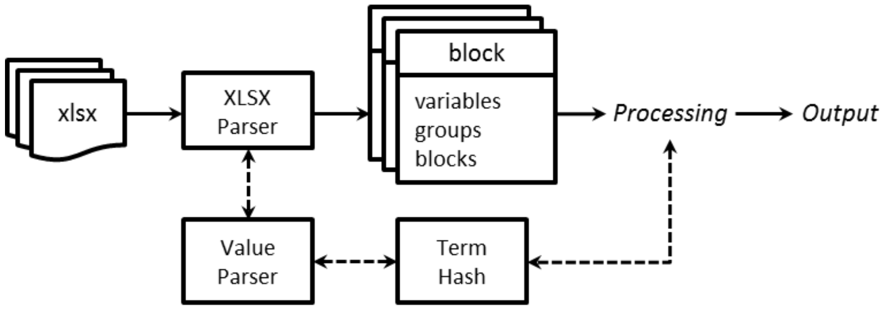


Fig. 5: Coversheet (xlsx) input, script processing block diagram, and output (generally SystemVerilog cover groups).

Processing of all coversheets, as in Fig. 5, traversed the entire directory tree in a depth-first fashion . A free XLSX reader [19] was employed in the XLSX Parser block to open and parse MS Excel workbooks. The config, mode, and cover variable worksheets were parsed first, and in that order, and then decorated to resolve reference bindings. Referenced variables, as those with a dollar-sign prefixed in TABLE 1 and TABLE 2, were considered “in-scope,” and thus resolved, if they had been defined in the same coversheet or any direct lineage parent block coversheet.

TABLE 5
VARIABLE / POINT RANGE GRAMMAR

start_variable_range	:=	term terms |	range	:=	**[** item **:** item **]**
start_point_range	:=	term terms | ***** |	item	:=	enum | value | var
terms	:=	**,** term	enum	:=	*SystemVerilog identifier*
term	:=	list | range | item	value	:=	*SystemVerilog value*
list	:=	**{** term list_items **}**	var	:=	$*string variable symbol name*
list_items	:=	**,** term list_items |			

Cover variables defined ranges while instantiated cover points employed those ranges, or subsets thereof. Cover variable range values, and their corresponding cover group point values, were read by the Perl-based Value Parser implementing the grammar in TABLE 5. Cover variable production rules start with “start_variable_range”, while cover point production rules start with “start_point_range”. Symbols in bold red are tokens within the syntax and treated as-is within the range, while the pipe ‘|’ indicates an alternative. Leaf value terms were items from the grammar that resolve to a numeric value or enumeration. Numbers conformed to either C- or SystemVerilog-syntax and could be arbitrary length bit or logic vectors, represented in Perl by [20] and [21]. Enumeration values are not specifically interpreted by the script, as shown in Listing 1 and Listing 2; their resolution was handled by the hardware compiler. Unique instances of terms across the cover model were ensured by liberal hashing, tying references in a globally accessible Term Hash [22]. For example, all uses of the number 16’h42 in the cover model resolved to the same term instance in script processing. The Term Hash, in Fig. 5, was used when creating terms.

Processing validated cover point value ranges by ensuring only valid terms from the corresponding variable were used. A cover point term was deemed valid when either it existed in the variable value range (3), or processing could ascertain it was a subset of a contiguous range (4).

$$\exists term \in variable : point\_term = term \tag{3}$$
$$\exists range \in variable : point\_term \subseteq range \tag{4}$$

Once the internal cover model was built, effectively modeled as a decorated abstract syntax tree, script processing manipulated it in several ways.

1. Variable/point transformation,
2. Configuration filtering of point and cross coverage, and
3. Minimal group cross coverage expansion.

An output from the model is a set of SystemVerilog cover groups for instantiation in the verification environment.

A. Variable/Point Transformation

Cover group point bins drive the overall adaptive functional coverage model. Script processing ensures only relevant coverage exists in the configuration. Then, by ensuring minimal sets for both point bins and their crosses, a cover group may be completely covered. Waivers are not required because no holes exist in the final customer-configuration SystemVerilog cover group. However, it is crucial that transformations on cover point bins do not disrupt the original coverage intent.

For config and mode type variables, transformations on their instantiated range value bins are applied immediately. All other variables are revisited, and transformations applied, during cover group processing. If a variable is unused in the model (declared but not instantiated as a cover point in any cover group), then it is not processed.

The variable and cover point value range contains an array of terms. Each term within the array may be a leaf term (number, enumeration, or contiguous number range), a list, or variable substitution (e.g., $Data). Within each list or variable substitution may be a nested list, variable substitution, or leaf term. There is no limit to the depth of nesting. However, to generate a SystemVerilog cover point, the entire array must be flattened until each term is a leaf term or a flat list (i.e., a list containing only leaf terms). Transformations are applied to the array until this state is achieved.

First, **substitution** recursively replaces all variable references in the array with actual terms (e.g., {$Data} $\Longrightarrow$ {[8'h00:8'hff]}). Variables may be substituted with a leaf term, a list, a variable reference, or any combination thereof. Thus, nesting generally increases following substitution. Second, all nested lists are **flattened** to their outermost list boundary and redundancies removed. A list of leaf terms is modeled in a SystemVerilog cover point as a single cover bin (e.g., coverpoint data { bins data_0 = { [8'h00 : 8'hff] }; }). Nested lists, therefore, expand the values to be covered in that single bin. For example, consider the list term {{1, 2}, 3, {4, 5}}. This term can be directly read as: either value in the list {1, 2}, or the value 3, or either value in the list {4, 5} satisfies this cover bin. Of course, that just means the cover bin is satisfied when any value 1–5 is observed. Algorithm 1 is employed to each term in the variable or cover point range array to flatten all nested lists.

Algorithm 1 Flatten nested lists in a term

```
    Require: term
 1  if term.Type = LIST then
 2    cur_lst ← new List;
 3    for i = 0 to length(term) do
 4      lst_elem ← Flatten(term[i]);
 5      if lst_elem = LIST then
 6        for j = 0 to length(lst_elem) do
 7          if lst_elem[j] ⊄ cur_lst then
 8            cur_lst.append(lst_elem[j])
 9          end if
10        end for
11      else if lst_elem ⊄ cur_lst then
12        cur_lst.append(lst_elem)
13      end if
14    end for
15    return hash(cur_lst)
16  else
17    return term
18  end if
```

PROOF. We must consider three types of terms: not a list, a flat list, and a nested list. If the term under examination is not a list, then it is already a leaf term because all variable references were removed during substitution. Therefore, the leaf term, a number, an enumeration, or a contiguous range is returned. A leaf term does not change. Suppose the term is already a flat list (i.e., each element of the list term is a leaf term). In this case, the recursive call in line 4 performs no action on each term as they are all leaf terms. The new list is hashed, yielding the same instance reference as the input, and returned. A flat list does not change. Now suppose the term is a nested list that contains a single flat list, such that the nest depth is one, and, possibly, other leaf terms. In this case, a new list is created in line 2 and the flat list term is recursively returned in line 4, unchanged. Then, in lines 5-10, each element of the flat list is appended to the new list, removing duplicates. The remaining leaf terms are added to the new list in lines 11-13, removing duplicates. The new list is hashed to ensure unique term instance and returned. Thus, lines 5-13 remove a single level of nesting. Since each term is recursively flattened in line 4, the algorithm concludes when all terms within the original list term have been reduced to leaf terms, leaving a single flat list. □

B. Configuration Filtering

Config cover variables mold a general functional coverage model into a customer-specific coverage implementation. For verification of a highly configurable IP, the config variables are critical in order to match the coverage model to the actual IP configuration.

Config variables are applied as band-pass filters mode variable ranges immediately after transformation. Only those mode variable bin values that exist in the config variable pass through the filter. In (5), m_var indicates a mode variable while c_var is its corresponding config variable. The resulting filtered mode variable instance contains only those terms that match the configuration.

$$\text{m\_var}_{filtered} \leftrightarrow \{ \forall i \in \text{m\_var} : i \rightarrow i \in \text{c\_var} \} \qquad (5)$$

Config variables also act as a band-pass filter for cross coverage. During cover group processing, config variables may be specified explicitly, as in TABLE 4, to further refine the group's coverage. If no terms exist after filtering the config variables values, then the whole cross is discarded for that customer-specific functional coverage model.

C. Minimal Cross Coverage Expansion

Cover group cross coverage is written with clarity in mind. Therefore, processing must perform two levels of expansion (1) explicitly indicate mode variables and (2) expand crosses into unique permutations. TABLE 3 shows an example of a leaf cover group with no mode variables. Automation add the mode variables as cover point columns and includes the entire variable range in each row. Next, each cross is built as a conjunction, as in (6).

$$cross_i \leftrightarrow \bigwedge_{j=1}^{\max\_var} range(\mathrm{var}_j, i) \neq \emptyset \rightarrow range(\mathrm{var}_j, i) \tag{6}$$

The *range* function returns the cover point value range (a set of one or more bins) for each specified variable var_j in column *j* for the *i*th row cross. The final result of (6) is a list, where each element is a conjunct of the cross.

Each element term in $cross_i$ may be a leaf term, a contiguous range, or a flat list (Algorithm 1 guarantees no nested lists). The next automation step expands $cross_i$ to one or more lists such that each list contains only leaf terms or contiguous ranges. The number of lists can $cross_i$ maximally expand to is equivalent to the product of each element term's list length, (7).

$$length\left(cross_{i_{expanded}}\right) = \prod_{j}^{length(cross_i)} length(\mathrm{term}_j) \tag{7}$$

However, to ensure a minimal number of crosses in a cover group, each expanded cross is hash such that only unique crosses are kept. Because each cross is also represented as a list, the list's instance reference is trivially used for hashing. Therefore, even when the same cover point cross is indicated in multiple rows, automation *only* generates a minimal set in the SystemVerilog cover group.

D. Minimal Cover Point Bins

Point coverage will likely indicate the same bin value in multiple rows of a cover group table. For example, in TABLE 4, the control cover point uses the bin value 1 for both *pkt_delim_cross* and *L0s_wake_rx_cross*. After minimal cross coverage expansion, section III-C, cover point bin values are collected for SystemVerilog cover points. For example, in Fig. 6, two variables are crossed in a cover group.

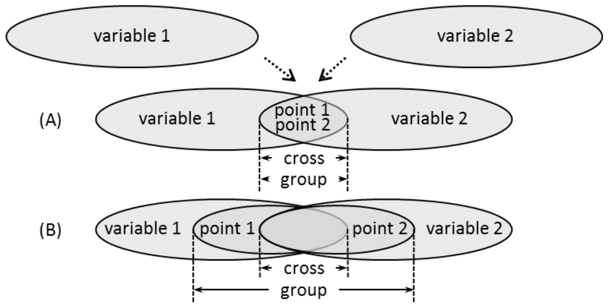


Fig. 6: Relationship of cover variables to cover points and within a group.

A minimal set of cover points collected from a minimal set of cross coverage is depicted in Fig. 6-(A). Here, the antecedent in (6) never evaluates to true because bin values from variable 1 are always crossed with bin values from variable 2. Only the cover point bins used in the cover group are collected. For Fig. 6-(B), however, some bin values from both variable 1 and variable 2 are crossed with the empty set (i.e., a cell in the row is left empty).

Each bin value is represented as a unique hashed term in the cover model. As such, a minimal set of cover point bins is collected trivially by inspecting the term reference and disallowing duplicates.

E. Cover Group Generation

Once both a minimal set of cross coverage and cover point bins have been achieved, the SystemVerilog cover group can be generated, as in Listing 1 and Listing 2.

V. Experience and Limitations

We have employed this functional coverage framework and script processing two PCI-Express projects, one as an IP and one as a subsystem incorporating other IP. Each project supports multiple simultaneous customer programs. The IP design implemented thirty unique customizations for the customer to select. For example, the IP may support 1, 2, 4, or 8 lanes of transmission, with or without IO-virtualization, and with or without low power support, including supporting a subset of low power options. Table 6 compares the total number of crosses in differing customer configurations, all PCIe end-point devices. Configuration 1 (C1) is a two lane device supporting L0s; configuration 2 is a four lane device supporting IO-virtualization and low power L1PM-substates. Configuration 3 is a four lane device with no low power. Example Perl script runtime and memory footprint required to process the super-set coverage model is indicated, as reference, per configuration executing on a shared Linux machine with 40 CPU cores and 264 GB of memory. The PCIe subsystem included a subset of the same options as the PCIe IP since it was instantiated within. A subset of the same cover groups were incorporated in the subsystem functional coverage model, too.

Table 6
Customer-specific model size comparisons and script processing metrics.

	Sample Config Variable Options			**Total SV**	**Script Processing**	
	Link Width	**IO-Virtualization**	**Low Power**	**Crosses**	**Time (s)**	**Mem (MB)**
C1	x2	no	off, L0s, L1	88,975	801	2,588
C2	x4	yes	off, L0s, L1, L1ss	126,496	1,182	3,632
C3	x4	no	off	128,514	1,175	3,689

We encountered three stumbling blocks, and one reporting opportunity, early in the IP program. First, it was difficult to accurately visualize the effect of cover point bin crosses, mode variables, and configuration variables on the generated SystemVerilog cover groups. However, because the coverage model is converted to an internal abstract syntax tree within the script, we were easily able to output, post transformations, a spreadsheet. We employed an XLSX writer [23] to generate the customer-specific coversheets, as per Fig. 7. This enabled a recursive refinement to the model: write a cover group coversheet, generate an output coversheet, refine.

Second, it was our tendency to cross *everything* and *everywhere*. While the resultant cover model contained valid bins, they were not always valid within the context of the cover group. Third, we declared too many root block mode variables each with term many terms. For each cross in each cover group, the mode variable has an exponential affect. These two issues resulted in a cross coverage explosion, one that could not be covered within a single project verification cycle. Therefore we developed some guidelines:

Guideline 1. Only cross variables that are actually necessary.
Guideline 2. Only cross term bins that are necessary to show that the desired testing scenario is covered.
Guideline 3. Minimize the number of mode variables and the number of bins in their terms.

After enacting these guidelines, our functional coverage models were much more feasible.

Finally, as we had already generated SystemVerilog cover groups and generated transformed XLSX coversheets, Fig. 7, we were able to also extend the script to generate a hierarchical Synopsys compatible Verification Planner XML file [8]. Each cover group defined in a coversheet was modified to include an instance path (multiple paths were supported), as reported by Synopsys Unified Reporting Generator (urg) [24]. Then, following regression, the Verification Planner XML file was back-annotated with functional coverage results to report the final cover score.

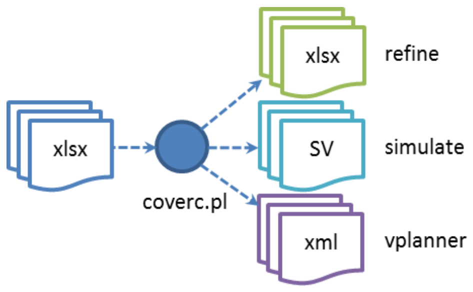


Fig. 7: Usage model for functional coverage automation flow.

VI. CONCLUSIONS

Effectively handling a functional coverage model for highly configurable IP designs is not straightforward. For each customer program, differing options lead to potentially disjoint coverage requirements. As such, it is easy end up with the functional coverage heavily relying on waivers to achieve goals. Instead, we offered an adaptive hierarchical model that, with a script, may be transformed upfront for each customer configuration. The resultant set of SystemVerilog cover groups do not require an extensive set of waivers as they have already been tailored to the customer's requirements. Furthermore, because our tool builds an internal model to operate on, it is possible to generate not only SystemVerilog but also the reporting structure (e.g., Synopsys VPlanner). Finally, once our flow was in place for our PCIe IP design, we were easily able to adapt the coverage model to a new customer program, and reuse with another project. Most importantly, we were able to juggle the coverage requirements in multiple customer programs simultaneously. Verification sign-off regarding functional coverage become straightforward.

REFERENCES

[1] PCI-SIG, *PCI Express Base Specification Revision 3.1,* 2013.

[2] J. Bergeron, Writing Testbenches using SystemVerilog, Springer Science+Business Media, 2006.

[3] P. James, Verification Plans: The Five-Day Verification Strategy for Modern Hardware Verification Languages, Kluwer Academic Publishers, 2004.

[4] L.-T. Wang, Y.-W. Chang and K.-T. Cheng, Electronic Design Automation, Morgan Kaufmann, 2009.

[5] A. Piziali, Functional Verification Coverage Measurement and Analysis, Springer, 2008.

[6] Cadence Design Systems, Inc., "Incisive© Enterprise Manager Verification Planning," 2014.

[7] Mentor Graphics Corp., "Questa© SIM Verification Management User's Manual," 2014.

[8] Synopsys, Inc., "Verification Planner User's Guide," 2014.

[9] C. I. Castro, M. Strum and J. C. Wang, "Automatic Generation of a Parameter-Domain-Based Functional Input Coverage Model," in *LATW2010, 11th Latin-American Test Workshop*, 2010.

[10] M. Teplitsky, A. Metodi and R. Azaria, "Coverage Driven Distribution of Constrained Random Stimuli," in *Design and Verification Conference (DVCon)*, USA, 2015.

[11] A. Krupp and W. Mueller, "Classification Trees for Random Tests and Functional Coverage," in *Design, Automation, and Test in Europe (DATE)*, 2006.

[12] S. Asaf, E. Marcus and A. Ziv, "Defining Coverage Views to Improve Functional Coverage Analysis," in *Proceedings of the 41st Design Automation Conference*, San Diego, 2004.

[13] H. Azatchi, L. Fournier, E. Marcus, S. Ur, A. Ziv and K. Zohar, "Advanced Analysis Techniques for Cross-Product Coverage," *IEE Transactions on Computing,* vol. 55, no. 11, pp. 1367-1379, 2006.

[14] D. Große, U. Kühne and R. Drechsler, "Analyzing Functional Coverage in Bounded Model Checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1305-1314, 2008.

[15] H.-H. Yeh and S.-L. Haung, "Automatic Constraint Generation for Guided Random Simulation," in *Design Automation Conference Asia and South Pacific (ASP-DAC)*, 2010.

[16] A.-C. Cheng, C.-C. Yen and J.-Y. Jou, "A Formal Method to Improve SystemVerilog Functional Coverage," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2012.

[17] L. Wall, "The Perl Programming Language," 2006.

[18] Microsoft Corporation, Inc., *Microsoft Excel,* 2010.

[19] D. Ovsyanko, *Spreadsheet::xlsx -- perl extension for reading MS Excell 2007 files, Rev 0.13,* 2010.

[20] P. J. Acklam, M. Biggar, I. Zakharevich and Tels, *Math::BigInt -- Arbitrary size integer/float math package, Rev 1.997,* 2011.

[21] T. Granlund and et al., *The GNU Multiple Precision Arithmetic Library,* 2012.

[22] G. Sarathy, *Tie::Refhash -- Use references as hash keys, Rev 1.32,* 2011.

[23] J. McNamara, *Excel::Writer::XLSX -- Create a new file in Excel 2007+ XLSX format, Rev 0.77,* 2014.

[24] Synopsys, Inc., "Coverage Technology User Guide," 2014.