Managing Highly Configurable Design and Verification

Jeremy Ridgeway

Broadcom, Inc.

Fort Collins, CO USA

www.broadcom.com

ABSTRACT

Significant challenges face highly configurable design and its verification. First, it may be difficult to tailor for verification a superset RTL to meet a specific customer's requirements. As a result, often a superset test bench is employed leading to delivery of RTL that has not been specifically verified. Second, reporting verification progress in a constrained random verification environment may be cumbersome. We present our approach tackling both these into a cohesive set of reports that accurately shows the current status of verification progress.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Let's be honest, managing highly configurable intellectual property (IP) hardware and verification is fraught with complication. No longer do we create a team of hardware and verification engineers to implement an ASIC. Today's market demands hardware to be designed with usage flexibility—often customers' use models are not known at the outset. Flexible hardware demands equally flexible verification architecture. The verification architecture is not limited to the test bench. It affects all processes from hardware design for verification through to sign-off for customer release. What metric qualifies the design under test (DUT)? In this paper we present our verification architecture and prevailing guidelines and lessons learned.

Overall guideline for verification:

> Verification architecture must allow organic growth. Hardcoding any value or approach will eventually be detrimental to some project schedule.

Guidelines:

1) Do not simulate hardware not delivered to the customer.
2) Identify top-level use cases, but do not be limited to them.

Note that we approach this presentation from a verification perspective. We are not hardware engineers, we are verification engineers. We are not providing novel hardware algorithms, just guidelines to aid in the possibility for success in a multiple-customer environment. Nonetheless, some or all our methods presented here have been employed in multiple IP and subsystem level projects over multiple, often simultaneous, customer projects.

In section 2. , we approach hardware design configurations. Being upfront and vocal about configurable hardware decisions will enable configuration verification flows within the architecture. Section 3. focuses on verification planning and reporting. These two sections make the backbone of our verification architecture. The remainder of the paper delves into deeper detail into each part of the verification architecture. Additional guidelines are marked **in bold**.

2. Design Configuration Considerations

To meet flexible usage demands, often an RTL superset is developed by the hardware design team that will support multiple target configurations. This approach has a clear advantage over dedicated RTL per customer in both design and maintenance resources. The RTL superset may be transformed into a specific customer configuration, often, in one of the following (or a combination of both) ways:

1) Compile-time macros, and/or
2) Pseudo- or constant-static run-time activation.

Consider, in Figure 1, some Capability X that is optionally available for the hardware design. For example, the PCI-Express (PCIe) standard allows an end point an optional dynamic power allocation (DPA) capability [1]. If the PCIe design also optionally supports DPA capability then DPA is a hardware *configuration*. Inclusion of the capability is dependent on customer requirements. That is, Capability X is *selected* when the customer demands the capability. Otherwise, Capability X is excluded from the overall customer use-model. Conversely, Capability X may be selected but never *activated* in the customer configuration. That is, Capability X is selected for inclusion into the design but (2), above, may be always on (constant-static activation) or off (constant-static deactivation). For DUT some configurations, we assume can the capability may only switch from on to off, and vice versa, through a drastic event, such as reset (pseudo-static activation).

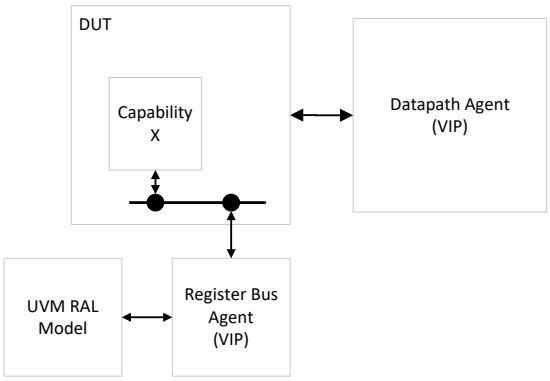


Figure 1: Functional test bench impact of some Capability X.

Depending on the logic size impact to the overall design, the hardware design team may opt solely for (2) and tie an input (or register field value) to constant-static value to ensure Capability X is always or never activated. Alternatively, compile-time `ifdef macro, (1), may fully include/exclude all logic associated with Capability X.[1]

The configurable hardware must be transformed to the customers' requirements. It may be possible to deliver superfluous hardware to the customer but at no point should unknown hardware be delivered. The size of the combined logic in a single System-on-a-chip (SOC) and the overall project schedule are the drivers for determining superfluous hardware acceptability. For example, if Capability X is not selected but its overall logic size is within limits for the IP then schedule may determine this superfluous configuration can be included, selected but never activated.

No verification of a highly configurable hardware design shall target a superset, Figure 2. Consider Capability X selected by `ifdef macro, option (1). If the customer does not select Capability X then its logic shall not be part of the verification for that customer. This implies that simulation for that customer must also not include Capability X. This implication holds if there exists no other way to show verification complete for that customer. As an alternative, functional coverage may be employed, in a constrained-random test bench, to support verification complete.

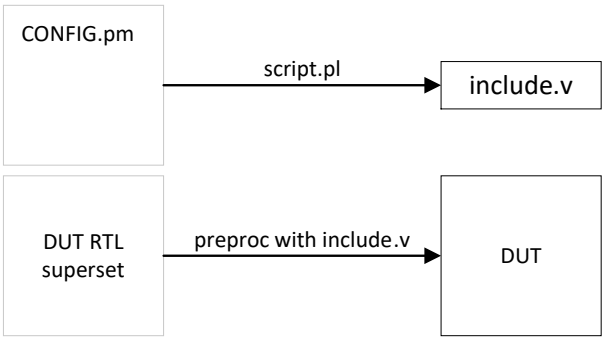


Figure 2: A configuration object (in perl) transforms the DUT superset to customer-specific.

[1] We have experienced both approaches exclusively and a combination of the two simultaneously.

2.1 Configuration Matrix

A configuration matrix clearly identifies all configurations within the hardware design. This configuration matrix is reviewed directly with the customer to select configurations. Configurations may be Boolean in nature (lending itself to `ifdef testing) or may be value oriented (lending itself to `macro instantiation). Table 1, is presented as a simplistic example. In our experience, the configuration matrix can include many rows. Project management and the customer identify the customer selection for all configurations in the matrix. Once agreed upon, this matrix becomes the Plan of Record for the customer's project. These configurations and values selected are used by the hardware design team to transform the DUT RTL superset to the customer design. The same configurations are used by verification to transform the test plan, functional coverage, test bench, regression, and reporting mechanisms to the customer design.

Table 1: Example Configuration Matrix.

Configuration	Customer Selection
Capability X	Supported
Reference clock frequencies (MHz)	100, 200

2.2 Configuration Object

Within the verification architecture, **all configurations for the customer start with a single *configuration object***. Automation is imperative for project success. Any other approach to hardware and verification management in a multi-customer situation will eventually prove untenable.[2] Therefore, it is most desirable for this configuration object to be coded in some scripting language. Note that it is always possible to transform the script configuration object into (System)Verilog. We have opted for Perl5 object-oriented scripting [2].

```
1   package CUSTOMER_A;
2     our $config = {
3       REFCLK_SEL => [100, 200], # Mhz, input to DUT or DUT register
4       DEFINES => {
5         DUT => [ "`define COMPANY_CAP_X",          # ifdef testing
6                  "`define COMPANY_INSTMACRO 4" ],  # macro instantiation
7         TB => [ "`define TESTMACRO_SUPPORT" ],     # ifdef testing
8       },
9     };
    1;
```

Code 1: Example customer configuration object written in Perl.

Configuration object members that transform the RTL superset should be coded as RTL for the DUT. These are limited to `ifdef testing or `macro instantiations, as in line 5, above. From Figure 2, scripting translates the DUT-specific DEFINES from the customer configuration object into an include.v file and, later, used to preprocess the DUT RTL superset. **Always prefix all macros for release to the customer with a company name or abbreviation**. This will help avoid compile-

[2] We have no scientific data to support this claim, only our own observations.

time macro name collision at the customer site. In contrast to the `defines, line 3 shows an example of an object member that requires a more complex structure. Here, the reference clock may be emitted as `define macro influencing the test bench, but it may also be used in other ways, only limited by the capabilities of the Perl language.

The hardware design team is responsible for the hardware and its delivery. As such, **the hardware design team shall own the content of the configuration object on a per customer basis**. The verification team is responsible for testing the hardware design as well as the verification architecture. As such, **the verification team shall own the structure of the configuration object**. Every configuration listed in the matrix shall have representation in the configuration object. The hardware design team will set that configuration object member as appropriate.

2.3 DUT Transformation

Figure 2 also indicates the DUT RTL superset is preprocessed with the include.v file and transformed into the DUT. We have used several different approaches to implement this transformation step: at compile-time and before compile-time via preprocessing.

2.3.1 Transformation at compile-time

The simplest method is to include the include.v file, with appropriate `define macros, along with the customer IP release, as in Code 2. The preprocess step will naturally occur during compilation for simulation or synthesis and unselected code removed from the hardware design.

```
1  `define COMPANY_CAP_X
2  `define COMPANY_INSTMACRO 4
```

Code 2: Generated include.v file used to implement the preprocessing set.

Two guidelines should be considered when providing a definitions file when part of the customer IP release.

a) Ensure the include.v file is listed within IP release compile-time file lists.

b) Consider encrypting the include.v file.

Even when using this simple approach, **never instruct the customer to specify the compile-time macros on their compilation command-line** (+define+MACRO). We have found that the customer's compilation command-line does not always update in between IP releases. As such, it may be easy to debug the customer's failing simulation only to find they had not specified the compile command-line correctly. With all the definitions in a file and part of the compile-time file lists provided to the customer this situation can likely be avoided. An additional security step of encrypting the include.v file would likely ensure the situation is avoided.

2.3.2 Transformation via preprocesing

A second approach to implement the preprocessing step from Figure 2 is to SystemVerilog preprocess the DUT RTL superset stripping out unselected RTL. Some simulation vendor tools may provide a path for preprocessing only, as well as third-party tools [3]. Similar to a gcc -E command (preprocess C code dumping output to the screen), the vendor's compilation program may preprocess files according to the compiler directives listed in the SystemVerilog standard [4].

Two potential issues with preprocessing:

a) Comments are usually removed, including synthesis or simulation hints, such as:

```
// synopsys translate on
```

b) Sometimes a few macros should remain as macros in the code, such as:

```
myreg <= `COMPANY_DELAY val
```

To address these issues we have implemented a light SystemVerilog preprocessor to handle the two types of macros listed in the configuration object.[3] We identify directly in the `define itself if this macro should *pass-through* the preprocessor. For example, the delay value from (b):

```
DEFINES => { DUT => [ …, "PASS_THRU `define COMPANY_DELAY 1" ] }.
```

This macro does transform the DUT RTL superset but the customer *should* have control over its value. Macros such as a delay used for simulation but not synthesis are the exception. After preprocessing, all other `define macros have been consumed (no longer exist in the hardware design) and logic replaced (`macro instantiation) or removed (`ifdef/`ifndef) or remains (`else). Importantly, all comments still exist in the RTL. Unlike the first approach, in section 2.3.1 , this approach requires the include.v file to *not* be listed in the compile-time file lists. The *only* macros that remain in the include_out.v file are those the customer should directly control for simulation or synthesis. Now it is impossible to encounter the debug situation mentioned previously: there are no macros in the hardware to define incorrectly.

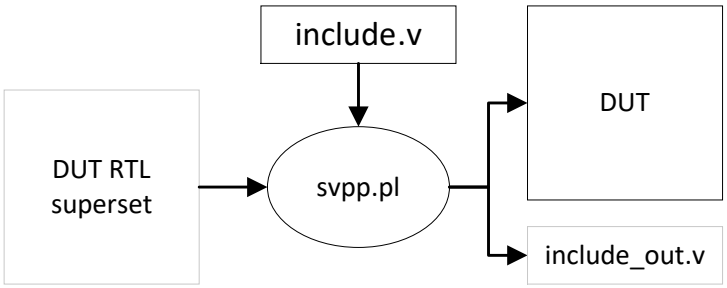


Figure 3: In-house light SystemVerilog preprocessor removes `define macros as required.

3. Test Plan and its Reporting

The first step for managing highly configurable verification is to define how the hardware design will be managed. In section 2. , we presented hardware configurations selected by `ifdef or other pseudo-static or constant-static methods. In this section we focus on the test plan.

[3] For obvious reasons we are not able to release this script, but we have seen other in-house flavors of this same script in other teams within Broadcom. The SystemVerilog Compiler directives chapter 22 is not very long [4].

3.1 Scenarios for Testing

From the DUT RTL superset technical manual and/or the industry standard, the verification team identifies and enumerates all testing scenarios that must be covered. The items listed in the test plan are focused on the specific scenario to be tested and not a full testing sequence. That is, a single scenario in the test plan does not precisely indicate the order of events of simulation (reset, initialize and select clock A, then do this other thing). Instead, the scenario describes, when the stimulus generation description is encountered, what behavior is expected from the DUT. A functionally passing scenario correctly takes the actions specified. For example, in Table 2, scenario ID DATA.CAP_X.1 describes a situation where the DUT is enabled to silently drop received data that has errors but then all data received is OK. This is a "good" test, no errors are encountered. For scenarios DATA.ERR.CAP_X1.1 and DATA.ERR.CAP_X1.2 the DUT receives error data and either drops it or interrupts firmware. These are a "good with errors" test intermixing good data with error scenarios. These scenarios could both occur in a single simulation or in different simulations.

Table 2: Example basic test plan common section.

Scenario ID	Feature	Subfeature	What to generate	What to expect
CLK.SEL.1	Clocks	Main clock selection	Randomize external clock generation to customer range.	No adverse situation in any simulation.
REG.CAP_X.1	REG	Capability X	Read default values	Values match register definition.
REG.CAP_X.2	REG	Capability X	Bit-bash fields	Access types match register definition.
DATA.CAP_X.1	Datapath	Capability X	Set control to DROP_AT_ERR then generate random good data scenarios.	All data received and correct.
DATA.ERR.CAP_X.1	Datapath Errors	Capability X	Set control to DROP_AT_ERR, then generate error scenario.	Data with error should be dropped by the DUT.
DATA.ERR.CAP_X.2	Datapath Errors	Capability X	Set control to INT_AT_ERR, then generate error scenario.	Data with error should cause DUT to interrupt.

The test plan, itself, allows for both directed and constrained random testing. Of course, there will be exceptions. For example, scenarios REG.CAP_X.1-2 are defined as directed tests. There is no need to construct these scenarios within the bounds of a "good" or "good with errors" random test as this is not the normal behavior of firmware. Conversely, scenario CLK.SEL.1 affects every single test – it is an overriding mode of operation of the DUT. We should be able to show register tests working properly in all selected clock modes. Finally, while the Capability X scenarios may be fully defined in the test plan, the CLK.SEL.1 scenario has no implementation until directed by a customer configuration.

When enumerated in a single common test plan, the scenario IDs become standardized across all customer projects regardless their implementation.[4] An example we have encountered is the register bus interface. When Capability X is selected in a customer configuration then REG.CAP_X.1-2 scenarios must be verified. However it is possible to perform that verification with an AHB register bus or an AXI register bus [5, 6]. The register bus selected is an `ifdef-style configuration of the DUT and will not change for that customer. Nonetheless, REG.CAP_X1-2 is a valid test for all

[4] This testing architecture is in the vein of the Standard Framework for Interpretation from [12].

projects that select Capability X and we can quickly determine pass or failure without knowing the underlying register bus technology. By abstracting the executed simulation from the testing scenario we can analyze scenarios more specifically and without requiring a specific run test.

3.2 Scenarios for Reporting

The valid testing scenarios for that customer configuration are mapped to run simulation (what test must be executed) and functional coverage. First, the valid test scenarios are directly affected by the customer `ifdef-style compile-time and constant-static run-time configurations. In either approach the testing scenario is either valid or invalid. Suppose, in Table 3, Customer B does not support Capability X. Then all testing scenarios associated with Capability X are not valid and are indicated as such in the test plan. For testing scenarios that are valid in the customer configuration, once they are available for regression they are also indicated as such. Considering only the scenarios listed in Table 3, verification for Customer A has implemented 3 of 4 tests; the err_capx test is not yet available. However, 4 of 6 scenarios are implemented. Therefore we would say that even if all tests passed in regression the verification for Customer A cannot exceed 66% coverage of the test plan from simulation perspective.

Table 3: Example basic test plan customer section; selection of scenarios for data collection and reporting.

Scenario ID	Test name	Customer A		Customer B	
		Valid	Regress	Valid	Regress
CLK.SEL.1	*All*	Y	Y	Y	Y
REG.CAP_X.1	reg_default_test	Y	Y	N	
REG.CAP_X.2	reg_bitbash_test	Y	Y	N	
DATA.CAP_X.1	base_test	Y	Y	N	
DATA.ERR.CAP_X.1	base_err_test, err_capx	Y		N	
DATA.ERR.CAP_X.2	base_err_test, err_capx	Y		N	

At the beginning of the overall project likely only the scenario details are known, as in Table 2. Once the verification environment implementation has begun then test names will start to be known. As customer configurations are added then scenario validity will be known. The point here is that the test plan is a live document. Its structure will remain the same but its contents will grow over time.

3.3 Verification Approach

While how to approach verification is a well known problem, we highlight how the test plan correlates in this section. Selection of the overall verification approach for a highly configurable design is dependent on the (initial) test plan. We identify three approaches to verification:

1) Directed,
2) Directed-random, and
3) Random.

In Table 3, the test name to be simulated in order to cover the scenario ID is listed. For example, REG.CAP_X.1 is covered when reg_default_test is simulated and passes. This is a directed test that always performs the exact same sequence of actions and checks. If it is possible to identify one or more directed tests for each scenario in the test plan, including their interrelationship, then the directed verification approach is sufficient. For Customer A, from Table 1, the REG.CAP_X.1 scenario requires two individual instances, one for when clock selection is 100MHz and one for

200MHz. Increasing the number of unique scenarios has a negative impact on directed verification efficacy. Furthermore, based on the presentation in Table 3, there is no obvious way to correlate the test name in the test plan and the test written. The test plan test name becomes a one-to-many relationship. While the regression snapshot (i.e., one regression) can give a clear picture of pass or failure of scenarios tested, it hinders deeper analysis (e.g., how many scenarios remain).

Directed-random verification expands the directed test by incorporating random variables in order to achieve more scenario coverage with fewer written tests. For scenario REG.CAP_X.1, randomizing the clock selection prior to executing the main body of the test would achieve the desired results over one or more regressions. Functional coverage may be implemented to prove this result but also may not be necessary as, in this case, both scenarios will, statistically, be covered.[5] With directed-random verification the test itself is still in direct control over the environment. Furthermore, now there is a one-to-one relationship between test name and written test. Scripting can quickly ascertain which tests have been simulated, their pass or failure rate, and which tests have not been simulated, and adjust the regression score accordingly. For Table 3, as mentioned in section 3.2 , scripting can limit the regression score to 66%, even if all simulated tests passed, because not all expected tests exist in the regression.

Random verification, by contrast, moves control of the verification environment from the simulated test to the random constraints in the environment itself. Scenario DATA.CAP_X1.1 is indicated only as a random test – the "good" base random test. Scenarios DATA.ERR.CAP_X1.1-2 show two tests, one is the "good with errors" base error random test, while the other is a directed tests. Simulating the "directed" test (presumably with the correct command-line options) ensures the generated scenario will be covered. The "directed" test in a random verification environment is not a true directed test. Instead of ensuring the same sequence of events during simulation, the "directed" test constrains only as many variables as necessary to achieve the generated scenario. Then, checkers verify the response.

Even fewer tests, than the directed-random approach, are required in order to cover the test plan scenarios in a constrained random verification environment. Additionally, constrained random verification will likely simulate scenarios within the legal bounds of the DUT but outside the bounds of the test plan. In this manner the test plan becomes a base level of testing for the IP. Those unplanned legal scenarios, especially when error injection is off or extremely limited, are where the power of the constrained random verification environment comes to the fore. However, there is a cost to the automation, especially in reporting. As unique scenarios are added to the test plan more and more of them will be covered only by the "good" or "good with errors" random test. Now we have a many-to-one relationship from scenario IDs to test names. Again automation may be hindered for deeper analysis unless accurate functional coverage is also incorporated into the test plan.

4. **Functional Coverage**

The functional coverage model for verification of highly configurable IP must, itself, be highly configurable. We employ the moldable and hierarchical superset functional coverage model as presented in [7]. In section 2. , we presented hardware configurations selected by `ifdef or other constant-static methods. These configurations are represented as configuration variables in our superset functional coverage as they do not change over the course of the customer program.

[5] Keeping the number of random variables low in the directed-random test enables statistics to work in our favor: the probability of *not* covering all cases is proportionately low.

Similar to the hardware logic, functional coverage covergroups, or cross bins within, are optionally selected based configurations. Also in section 2. , we presented hardware configurations selected by pseudo-static methods at run-time. These configurations are represented both as configuration variables and mode of operation variables, or simply mode variables, in our superset functional coverage. For example, referring to the customer requirement matrix in Table 1, Capability X support is modeled only as a configuration variable. Either the covergroup, or collection of cross bins, exist in the functional coverage model for reporting scenarios hit, or they do not. However, reference clock selection may be modeled as a combination of configuration and mode variables.

Table 4: Functional coverage model configuration and mode variables.

Name	Range	Signal	Description
C_CAP_X	0,1		Capability X support.
C_REFCLK_SEL	100, 125, 200, 250, 500		Supported clock frequencies
M_REFCLK_SEL	$C_REFCLK_SEL	CFG::REFCLK_SEL	Runtime clock selection
CAP_X_CTRL	DROP_AT_ERR, INT_AT_ERR		Capability X control activation
CAP_X_ERR_IND	0,1		Capability X logic has seen an error

The moldable functional coverage model does not directly define cover points and their bins. Instead, it declares coverage variables with their full potential range of values. These cover variables are then instantiated within a cover group and SystemVerilog code generated as coverpoints. Furthermore, configuration variables are consumed by the coverage compiler script, much like `macros are consumed by the SystemVerilog preprocessor. That is, the configuration variables are used internal to the superset functional coverage model specification to mold the covergroup and its cross bins such that the generated SystemVerilog contains *only* those bins valid in the configuration. Figure 4 shows the general use model for the functional coverage compiler employed in our verification architecture. The superset functional coverage model is defined as a collection of Microsoft Excel XLSX workbooks [8]. An input configuration file for the coverage compiler, coverc.pl, defines configuration variable overrides such that the resultant SystemVerilog code has been molded to the customer configuration.

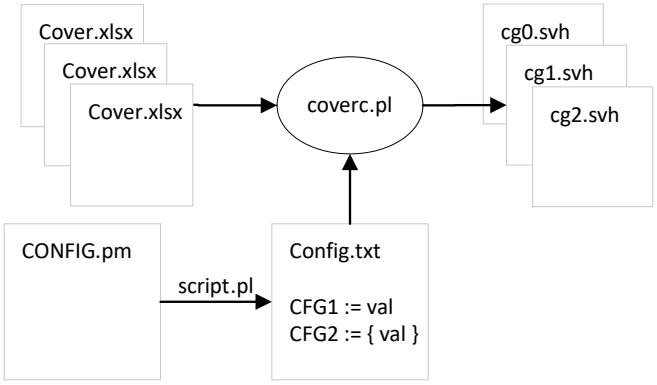


Figure 4: General use model for molding the superset functional coverage model.

By definition from [7], configuration variables are defined in the MS Excel workbook "config" spreadsheet and have names beginning with "C_" (e.g., C_REFCLK_SEL). Mode variables are defined in a "mode" spreadsheet and have corresponding names beginning with "M_" (e.g., M_REFCLK_SEL). Then, when the script identifies a mode variable with corresponding configuration variable it rectifies the range of the mode to the configuration variable. For example, the default expansion of

M_REFLCK_SEL from Table 1 is shown in Code 3.

```
1  // M_REFCLK_SEL := $C_REFCLK_SEL := 100, 125, 200, 250, 500; default
2  coverpoint M_REFCLK_SEL {
3    bins M_REFCLK_SEL_0 = { 100; }
4    bins M_REFCLK_SEL_1 = { 125; }
5    bins M_REFCLK_SEL_2 = { 200; }
6    bins M_REFCLK_SEL_3 = { 250; }
7    bins M_REFCLK_SEL_4 = { 500; }
8  }
```

Code 3: M_REFCLK_SEL default SystemVerilog coverpoint implementation.

However, refer to the configuration object for Customer A in Code 1. REFCLK_SEL configuration only supports 100MHz and 200MHz clock frequencies. As shown in Figure 4, the automation generates a configuration file for input into coverc.pl. For Customer A, the Config.txt file would indicate C_REFCLK_SEL := 100, 200. Based on this command-line override, the coverage compiler generates the expansion of M_REFCLK_SEL coverpoint as shown in Code 4.

```
1  // M_REFCLK_SEL := cmd-line override:= 100, 200; Customer A configuration
2  coverpoint M_REFCLK_SEL {
3    bins M_REFCLK_SEL_0 = { 100; }
4    bins M_REFCLK_SEL_1 = { 200; }
5  }
```

Code 4: M_REFCLK_SEL SystemVerilog coverpoint implementation for Customer A.

Referring to the test plan in Table 2, scenario DATA.ERR.CAP_X.1 indicates the situation where the activated Capability X control register is set to DROP_AT_ERR and error has been seen by the DUT. This covergroup should be sampled at the monitor step because it is reporting that situation has arisen where the DUT was configured for errors and an error was indicated. The verification of data actually dropped must be performed by some environment checker and *not* the functional cover group. Table 5 shows an example covergroup implementation while Table 6, shows the covergroup correlation back to the testing scenario. The full instance path must be provided in the correlation to properly collect functional coverage data. Just as the more than one test name may correlate to the scenario ID, so too can more than one coverage metric. Furthermore, while we are presenting only one type of coverage metric, the cover group instance, we may correlate into the covergroup as necessary (cover point or cross bin). These instance paths are taken as-is for generated verification plan back-annotation.

Table 5: Example covergroup for DATA.ERR.CAP_X.1 testing scenario.

Name **Instance path**	DATA_ERR_CAP_X_1_cg top::cap_x_agent_fcov::DATA_ERR_CAP_X1_cg		
Points	CAP_X_CTRL	CAP_X_ERR_IND	M_REFCLK_SEL
drop_err_indicated	DROP_AT_ERR	1	*

As functional coverage becomes available, the test plan may be updated, as in Table 6, to indicate function coverage correlation with testing scenarios. A passing scenario is one where all instances of its test name pass in regression and all its functional coverage metrics are completely covered. For example, suppose the base_err_test was regressed over 100 simulation instances and 80 of them passed. Also suppose that the covergroup for DATA.ERR.CAP_X.1 hit in the 100 MHz clock cross scenario only for Customer A. Because there exists two cover points for M_REFCLK_SEL for

Customer A, this represents a 50% coverage rate, or 1 of 2 cross scenarios have been covered. The scenario DATA.ERR.CAP_X.1 failed with a score of:

DATA.ERR.CAP_X.1 = (passing rate) * (coverage rate) = (80%) * (50%) = 40% regression score.

Table 6 indicates for Customer A one scenario has yet to be ready for regression. Assuming (a) all other tests passed, and (b) all other covergroups achieved full coverage (both clock modes were hit in other implemented covergroups), then the regression score can be calculated as:

Regression score = AVERAGE(scenario_score) * (implementation ratio)
= AVERAGE(CLK.SEL.1_score, REG.CAP_X.1_score, REG.CAP_X.2_score, DATA.CAP_X.1_score, DATA.ERR.CAP_X.1_score) * (5/6)
= AVERAGE(85%, 100%, 100%, 100%, 40%) * (5/6)
= 85% * (5/6)
= 70.83%.

Because the regression score is less than 100% there exists a verification failure (regression test failed) and/or a testing gap (implemented ratio and reported coverage). For this regression there's all three:

a) 20% of the base_err_test simulation instances failed,

b) Functional cover group for DATA.ERR.CAP_X.2, if implemented reports 0% coverage in simulation, and

c) DATA.ERR.CAP_X.2 testing scenario is not yet available.

As these issues are rectified, over time, the regression score should trend towards 100%.

Table 6: Example test plan with functional coverage correlated per scenario.

Scenario ID	Test name	Coverage Metrics	Customer A		Customer B	
			Valid	Regr	Valid	Regr
CLK.SEL.1	*All*	group: top::clk_cg	Y	Y	Y	Y
REG.CAP_X.1	reg_default_test	group: top::clk_cg	Y	Y	N	
REG.CAP_X.2	reg_bitbash_test	group: top::clk_cg	Y	Y	N	
DATA.CAP_X.1	base_test	group: top::cap_x_agent_fcov:: DATA_CAP_X1_cg	Y	Y	N	
DATA.ERR.CAP_X.1	base_err_test, err_capx	group: top::cap_x_agent_fcov:: DATA_ERR_CAP_X1_cg	Y	Y	N	
DATA.ERR.CAP_X.2	base_err_test, err_capx	group: top::cap_x_agent_fcov:: DATA_ERR_CAP_X2_cg	Y		N	

Just as the DUT is implemented as RTL superset, and the test plan is implemented as scenario superset, the functional coverage model is also implemented as covergroup superset. The single configuration object, written in Perl for our projects, transforms the functional coverage to customer configuration. Additionally, the test bench itself may conform to the customer configuration to ensure proper verification.

5. **Constrained Random Test Bench**

The constrained random verification environment should transform in the same manner as the DUT to verify the customer configuration. In section 2. , we presented customer configurations, specified through the single configuration object, in Perl, selecting static configurations via `ifdef compile-time or constant-static run-time selection. The test bench itself should transform in the

same manner at compile-time or constant-static run-time selection to verify the customer configuration DUT. Whereas for the DUT we presented multiple approaches to handling, and possibly stripping, compile-time macros, for the test bench this is not necessary.[6] Additionally, we presented pseudo-static run-time activations as mode of operation configurations. A top environment configuration object should contain and randomize activation values at the beginning of simulation to cover mode of operation configurations. In this section we cover some higher level topics specific to managing the testing and reporting for the highly configurable IP.

The test bench for highly configurable IP should focus only on constrained random verification, as described in section 3.3 , with some directed testing via confining constraints. Given the scope of a multi-customer situation and their corresponding mutually exclusive configurations it should be obvious that selecting and activating UVCs and agents is far easier than porting directed or directed-random testing from one customer project to the next. When an issue is found affecting a commonly-selected UVC then the fix may be applied in one code location rather than each projects' directed-random test.

In this vein, we have structured our project code vault in the following manner.

common/tb	Common test bench components: UVCs and/or agents.
common/tests	Common test base classes for both "good" and "good with errors" testing.
common/scripts	Project wide scripts.
common/coverage	Superset functional coverage model.
common/docs/testplan.xlsx	Superset test plan.
customer_a/tb	Customer A test bench class extensions and instantiations thereof.
customer_a/tests	Customer A test class extensions and instantiations thereof.
customer_a/scripts	Customer A specific scripts, including CONFIG.pm configuration object.
customer_a/docs	Customer A internal and releasable documents.

Each customer project starts with a configuration object, written in Perl and residing at customer/scripts/CONFIG.pm. We assume that in addition to simply instantiating common UVCs and tests some of those classes will require extension for that customer. For example, if Customer A project not only includes agents for Capability X but also must instrument some of its options, then Customer A/tb/uvc_cap_x directory would extend the necessary classes from common/tb/uvc_cap_x directory. In this manner the bulk of development during the course of the project occurs in the common directories. The goal is for the customer specific directories to require an initial setup at project start then very little development as the customer project progresses.

[6] If the test bench is released to the customer, then preprocessing the test bench may be desired but we have not had that requirement from any customer.

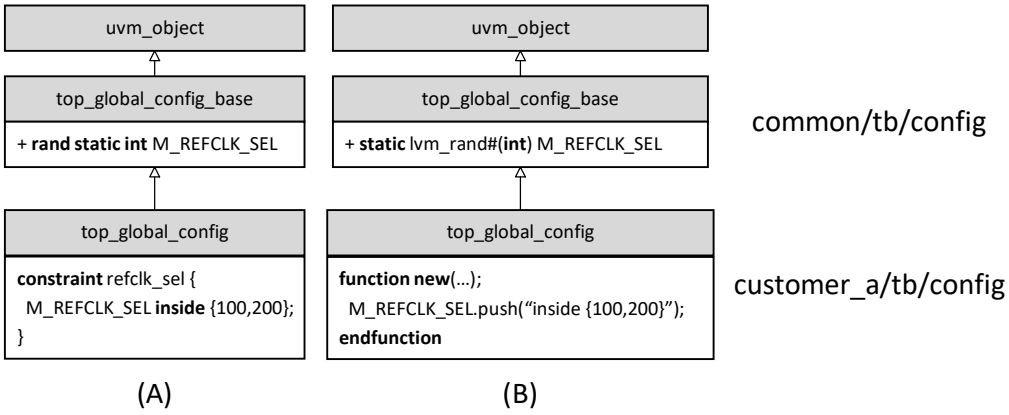


Figure 5: Top global configuration object structure defined in common directory, extended to customer project.

Regarding mode of operation configurations, refer to Figure 5 (A), the top environment configuration object is also be defined in some common/tb/config directory. This class provides the structure for the simulation-specific configuration object, containing all configuration members: compile-time specified or run-time specified. The constraints on those members are defined a customer_a/tb/config directory as class extension. For example, Table 4 shows the M_REFCLK_SEL functional coverage mode variable bound to the specified signal path CFG::REFCLK_SEL. The config object in the common/tb/config directory may contain both instance-specific (allowing for multiple top environment instantiations)[7] and static members. REFCLK_SEL, presumably, affects all DUT and test bench components and, thus, is represented as static random variable, unconstrained. However, the customer_a/tb/config directory configuration object class extension will apply the constraint valid for that customer.

```
1   // Common directory global configuration base class
2   class top_global_config_base extends uvm_object;
3     static lvm_rand#(int) M_REFCLK_SEL;
4     function new(string name = "top_global_config_base");
5       super.new(name);
6       M_REFCLK_SEL = new("M_REFCLK_SEL", this);
7     endfunction
8   endclass
9
10  // Customer-specicific directory global configuration base class
11  class top_global_config extends top_global_config_base;
12    function new(string name = "top_global_config");
13      super.new(name);
14      M_REFCLK_SEL.push("inside {100,200}");
15    endfunction
16  endclass
```

Code 5: Top global config classes defining the structure and default constraints on mode of operation configurations.

[7] Consider an IP that, for some customer, is actually instantiated twice, bundled together and delivered as a new IP. The test bench for one instance may be duplicated when architected with this kind of portability in mind. We have had this exact requirement for our DUT and in our test bench.

As shown in Figure 5 (B), our test benches employ the lvm_rand random object container class for mode of operation configurations [9, 10]. The advantage here is the container class automatically enables command-line modification of the constraint. For example, given the instantiation of the lvm_rand class, in Code 5, the following simulations commands may be used to choose a different constraint or confine the constraint.

```
> runsim +M_REFCLK_SEL='dist { 100:=10, 200:=1 }' ...
> runsim +M_REFCLK_SEL=100 ...
```

When not supplied, the M_REFCLK_SEL random variable has uniform probability on the customer valid values.

Finally, tying it all together into single compilation, Figure 6 shows our flow from the superset to the customer configuration compilation and simulation. As indicated, we are targeting Synopsys VCS toolset [11].

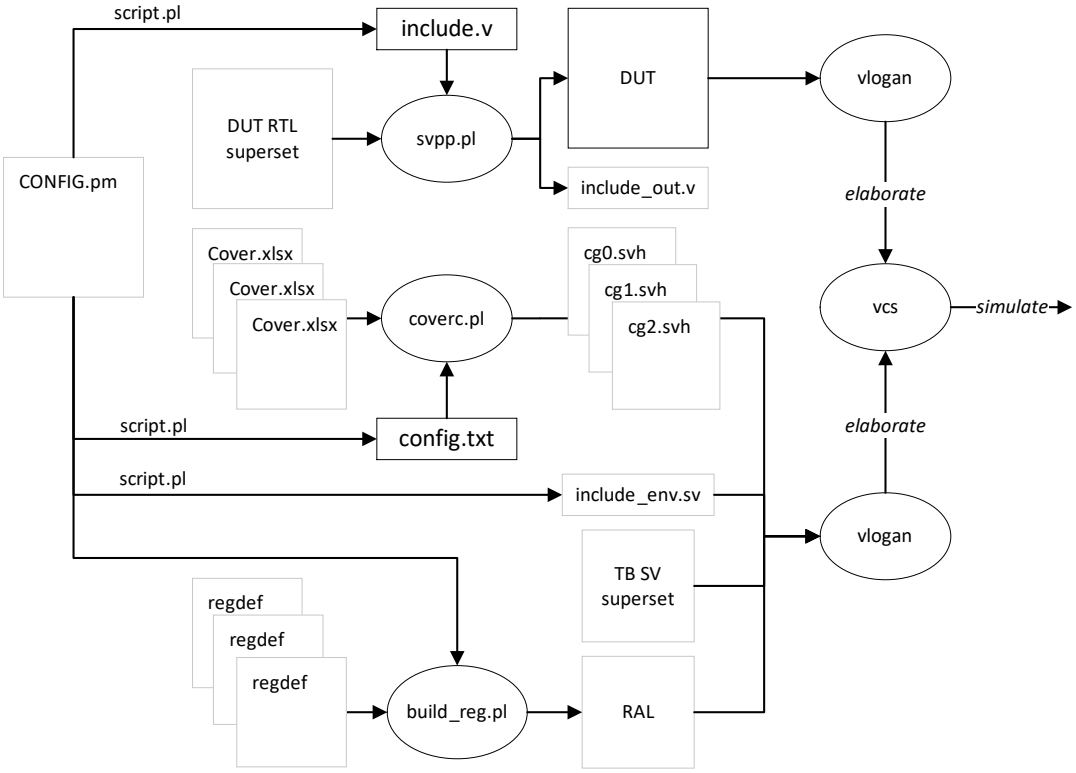


Figure 6: From customer configuration to simulation.

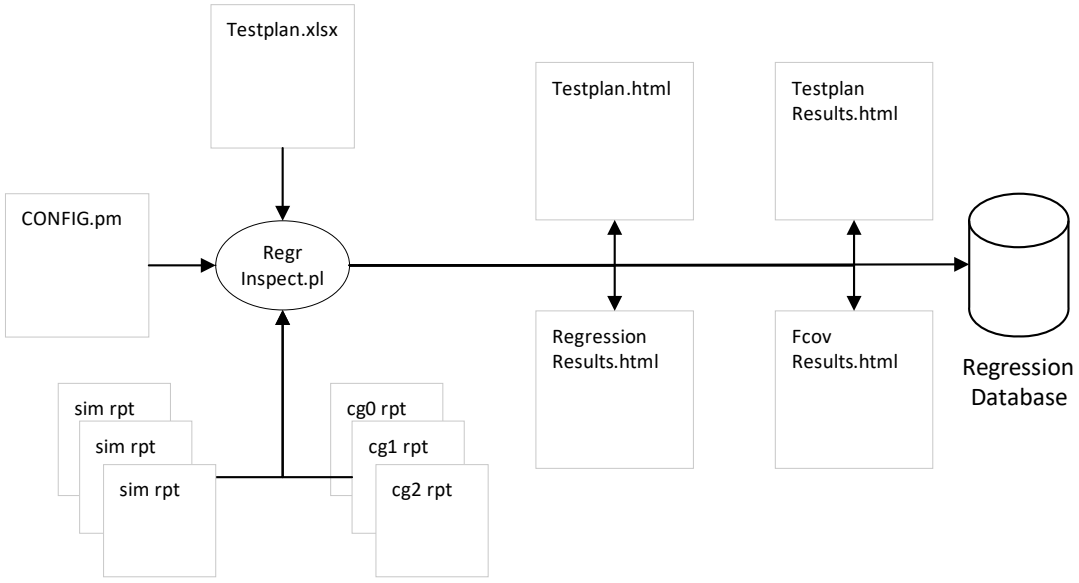


Figure 7: Post-regression reporting mechanisms.

6. Verification Progress

Verification reporting may be the most import component of the verification management architecture. This step capitalizes on the previous sections' work to collect the data and produce reports. Figure 7 presents the data input to the reporting script and the reports generated and Table 7 indicates the reports' dependencies and tools involved.

Table 7: Verification Reports and their dependencies.

Report	Dependencies	Tools
Testplan.html	Testplan.xlsx	In house script only
RegressionResults.html	All simulation reports (sim_rpt)	In house script only
FCovResults.html	All functional coverage (cg_rpt)	Synopsys URG, HVP [11], and in house program
TestplanResults.html	All reports	All the above

The TestplanResults.html report is the primary report to indicate current verification status. An example summary table for the testplan results is shown in Figure 8 (test scenario == testcase). The first section, "Testplan Results", shows brief accounting of the number of testing scenarios collected while parsing the Testplan.xlsx file for this customer configuration. The overall functional coverage score was a respectable 97% (although functional coverage metrics continue to be added daily). The "Regression Results" section showed a passing rate of 93%, but there are some issues still to be resolved. First one test was "missed." This test was marked in the Testplan.xlsx as in regression but was not seen by the inspection script to have simulated. Additionally, four tests are still to be implemented to cover an additional 10 test scenarios. Therefore the relative number test scenarios covered is:

128 test scenarios in regression * 93.161% regression passing = 119.246 relative test scenarios

Combining the relative number of test scenarios covered with the cover score results score of 83%. Also note the Testplan.html, functional coverage model and coverage results links available.

Summary

Testplan Results	
Total Sheets	10
Total Features	27
Total Subfeatures	81
Total Test Scenarios	139
Functional Coverage Results	
Total Metrics	27
Func Cov Score	97.607%
Regression Results	
Regressed	17 tests covering 128 testcases over 2091 sim instances 93.161% passing
Missed	1 test covering 1 testcase
Need	4 tests covering 10 testcases
Yet to-be-determined	some tests to cover 0 testcases
Covered tests	128 * 93.161% = 119.246 testcases
Testplan tests	139 testcases
Score	(119.246 / 139) * 0.97607 = 83.736%

Regression Results	Regression_results.html
Testplan Detail	Testplan.html
Functional Coverage Model	coverage/Cover_out.html
Functional Coverage Results	coverage_report_functional/dashboard.html
Code Coverage Results	coverage_report_code/dashboard.html

Figure 8: TestplanResults.html summary table.

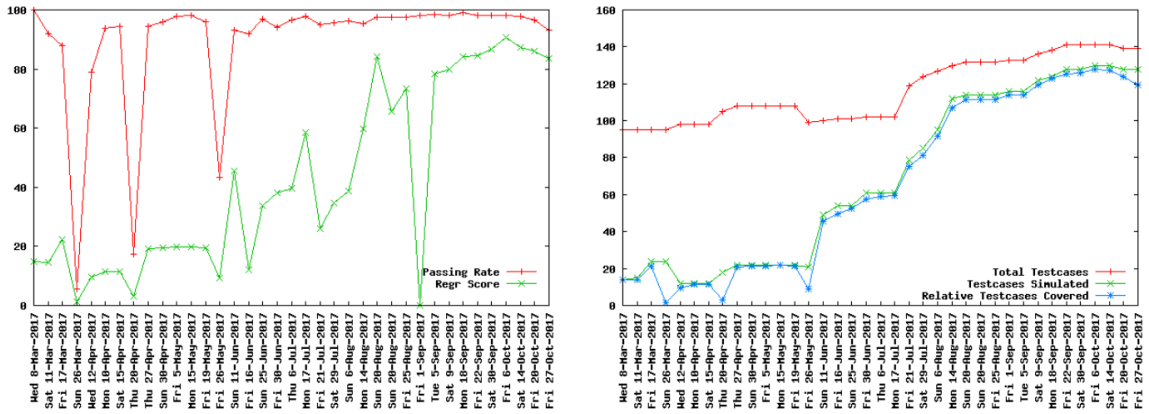
In Figure 9, the results for a single test scenario are indicated. This scenario is similar to the scenario in Table 2 of the same name. The functional coverage here was met at 100% but the test passing rate is inadequate. As such, the scenario is not covered in our verification architecture. Note that because this scenario affects *all tests* the number of instances and passing rate match the regression results themselves.

Test Scenario Results

	Test Scenario ID/Test Name	Instances	Rate	Result
CLK.SEL.1				FAIL fcov_met
1	all_tests	2091	93.161%	FAIL
1	group: top::pss_autoclk_fcov::pss_autoclk_cg		100.00%	fcov_met

Figure 9: Once testplan scenario showing failure due to inadequate simulation result even though functional coverage was met.

The results indicated in these diagrams are charted over the course of the project utilizing the regression database. Whereas the regression results can only provide a current snapshot we also chart the number of relative test cases covered. In Figure 10 (A), the regression percent-passing snapshot over time is charted. This only provides information about how well regression passing, not the verification status. However, Figure 10 (B) charts the relative test cases covered versus the total number of tests cases. This should always increase over time until the relative number matches the total number. In this we have all tests and functional coverage in place and they are all passing.



(A) (B)

Figure 10: Progress: (A) regression results snapshots over time; (B) progress towards relative test cases == total test cases.

7. Experience

We have incorporated this full verification architecture in our current PCIe subsystem project. Each customer selects the configurations from the master configuration matrix maintained by management and hardware designers. The configuration object is coded directly from the matrix selections. We employ a script architecture, in Perl5, to allow for more reuse between each of the script.pl's in the diagrams. This same script enables some parallelism within a single step in the flow and between steps in the flow. For example, when executing the RegrInspect.pl script from Figure 7, each of the simulation directories must be entered to determine the self-check test status. We parallelize this within the step. Also, generating functional coverage, Figure 4, may occur in parallel with DUT preprocessing or other steps. The script framework handles forking each of these scripts.

We have encountered challenges along the way, predominately refocusing our work from a customer design specific to a superset specific development process. This takes some realignment on thinking for the verification environment (although this is probably the easiest to picture), functional coverage and test plan. However, the dividends paid are from management having a clear understanding where each project resides in the schedule.

The Synopsys verification planner back-annotation becomes difficult to work with in this architecture. After back-annotation the hvp tool writes back to an XML file rather than an XLSX file. Publicly available Perl tools to convert back to XLSX are hard to come by (i.e., we haven't found any reliable tools). As such, we have replaced Synopsys hvp with an in-house program to connect to the unified coverage API (UCAPI) library. This becomes more difficult to work with but since this program has been in place we have had little trouble.

8. Conclusions

Again, let's be honest, managing a highly configurable design IP and corresponding verification project is difficult. Our verification architecture has evolved over several years and customer project generations. We are at a point to take full advantage of this architecture and provide accurate reporting on the overall progress of the project. Importantly, we can combine a test plan, coverage, and individual simulations from a full constrained random verification environment and report intelligently.

9. References

[1] PCI-SIG, PCI Express Base Specification Revision 3.0, 2010.

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

[3] W. Snyder, "vppreproc - Preprocess verilog code using verilog-perl," 2003.

[4] IEEE Computer Society, SystemVerilog, 2012.

[5] ARM, AMBA Specification, 1999.

[6] ARM, AMBA AXI and ACE Protocol Specification, 2011.

[7] J. Ridgeway, "Molding Functional Coverage for Highly Configurable IP," in *Design and Verification Conference (DVCON)*, San Jose, 2016.

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

[9] J. Ridgeway, "Interchangeable SystemVerilog Random Constraints," in *Synopsys User Group (SNUG)*, San Jose, 2014.

[10] J. Ridgeway, "Engineered SystemVerilog Constraints," in *Design and Verification Conference (DVCON)*, San Jose, 2015.

[11] Synopsys, Inc., "VCS MX/VCS MXi User Guide," 2017.

[12] A. Wiemann, Standardized Functional Verification, Springer, 2008.