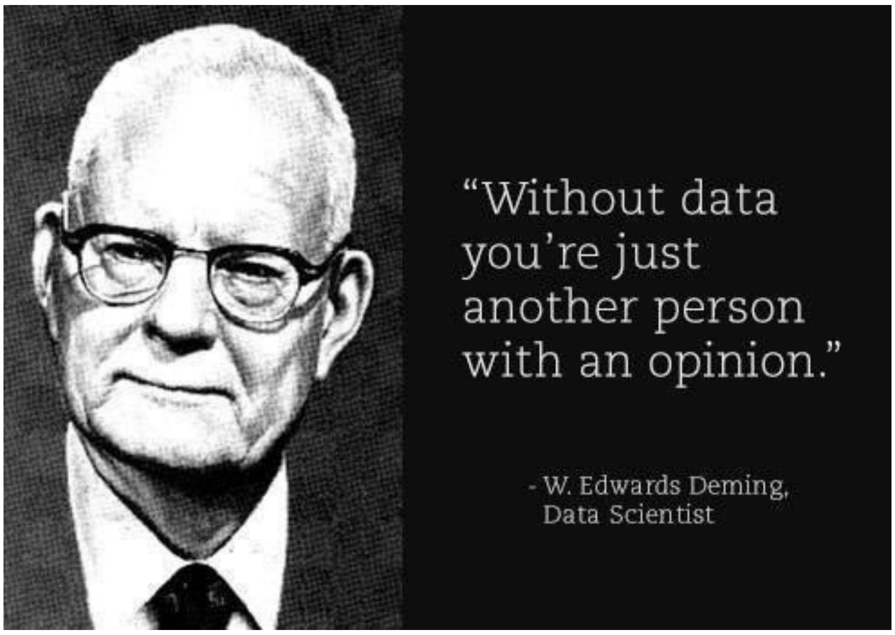
VerifSudha

# Constrained Random Stimulus quality analysis using Whitebox Functional and Statistical coverage

**Author:** Anand Shirahatti
(anand@verifsudha.com)

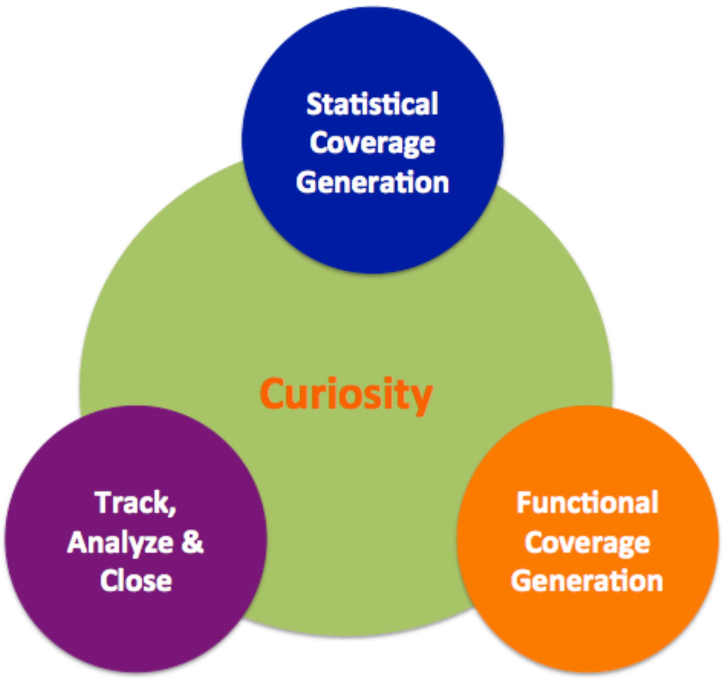

"Without data you're just another person with an opinion."
- W. Edwards Deming, Data Scientist

## 1 Curiosity: Stimulus quality analysis tool

We at VerifSudha have created a tool called curiosity by exploring various facets of how to achieve functional verification quality. You can read more about our journey through 100+ blogs dedicated to functional verification quality on our website (www.verifsudha.com).

Curiosity can help you throughout the verification process but it can make significant difference in helping you close on that last 20 % effectively. There by reduces the possibility of missing on those hard to find bugs.

As a by-product it will also help you optimize your regressions saving you time, compute farm resources, engineering resources and simulator licenses.


Statistical Coverage Generation
Curiosity
Track, Analyze & Close
Functional Coverage Generation

*If you can't measure it you can't improve it.*
*- Peter Drucker*

*Figure: Key capabilities of curiosity tool*

Curiosity helps you improve functional verification quality by measuring its effectiveness using:

- **Statistical coverage:** This is a unique feature offered by the tool. This allows you to find out stimulus is doing its job sufficiently and relative distributions are inline with your current project priorities
- **Functional Coverage:** With advanced built-in functional coverage models and object oriented functional coverage based reuse you can quickly generate the functional coverage for wide variety of

requirements. This helps you setup the coverage targets for discovered holes during exploration

- **Closure:** Tool supports flow where coverage planning, code generation and coverage results analysis can be done in single view minimizing the possibility of items falling through the cracks during the transformation. This ensures once holes is captured in plan its bound to get closed

Curiosity is non-invasive. It does not affect your test bench or RTL. It is agnostic to verification methodology. Even Verilog only legacy test benches can utilize all the capabilities. Any test bench running on simulator with SystemVerilog support can utilize all the capabilities of tool.

## 2 UART: Case study

To demonstrate some of its capabilities we did a simple case study using simple UART test bench. Complete UVM test bench of UART is selected from the mentor graphic's verification academy. This test bench is released under apache license. Anyone with verification academy account can download it from their website.

### 2.1 UART: Quick Refresher

You can skip this section if you are already familiar with UART.

UART stands for Universal Asynchronous Receiver-Transmitter. It's a simple full duplex serial communication protocol. The speed of data transfer is governed by baud rate which indicates the number bits per second.

Protocol is very simple. Protocol data transfer unit is a character. Number of data bits in character can be programmed to 5-bits, 6-bits, 7-bits or 8-bits.
All data transfers start with a start bit represented by line being pulled low for duration of one bit. There on the pre-programmed number of data bits are transmitted. Followed by data bits an optional parity bit is transferred. Parity can be programmed to be even or odd. Followed by optional parity is the stop bit, which again pulls the line high. The number of stop bits can be programmed to either one or two.

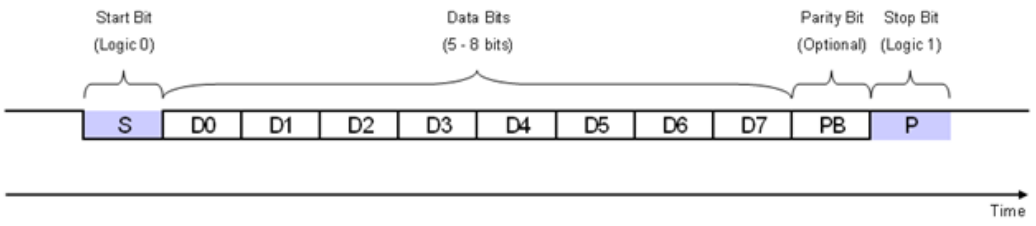


*Figure: UART packet format*

Typical UART design contains a baud rate generator, transmit channel, receive channel and control & status interface. Baud rate generator generates the clock for controlling the rate of data transfer. A common baud rate is selected from set of pre-defined baud rates supported.

Transmit and receive channel are implemented as a simple FSM for transmission and reception of start bit, data bits and stop bits. Transmit and receive side also include the FIFOs to hold multiple characters. This allows software application to program multiple characters to be transmitted or received.

Control and status block provides the ability to program various configurations discussed above and provides the status. Software programming interface for characters can be handled by software using either interrupt or polling operation.

Early days UART were used in the modems. So they provided the additional pins to interact with the modem mainly used for the flow control.

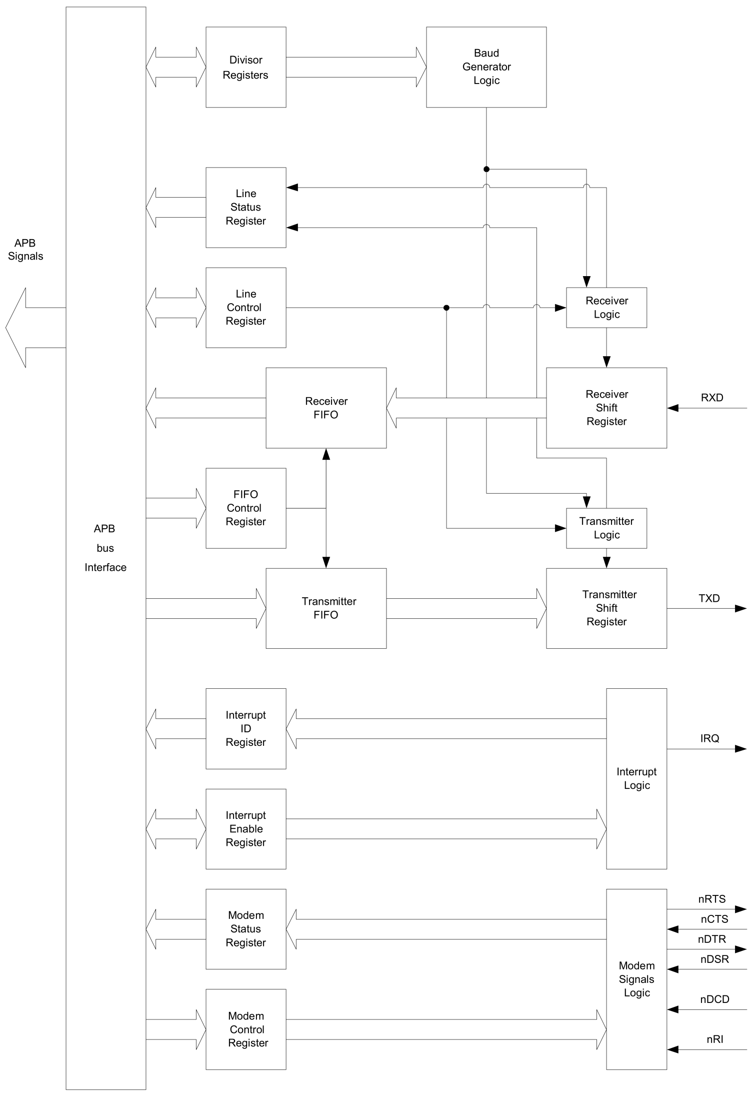


*Figure: UART Block diagram*

Example test bench is typical UVM test bench. Test bench is bundled with 7 tests to test various functionalities.

Please note that this test bench and design are not real life example but the methods demonstrated are applicable to real life designs as well.

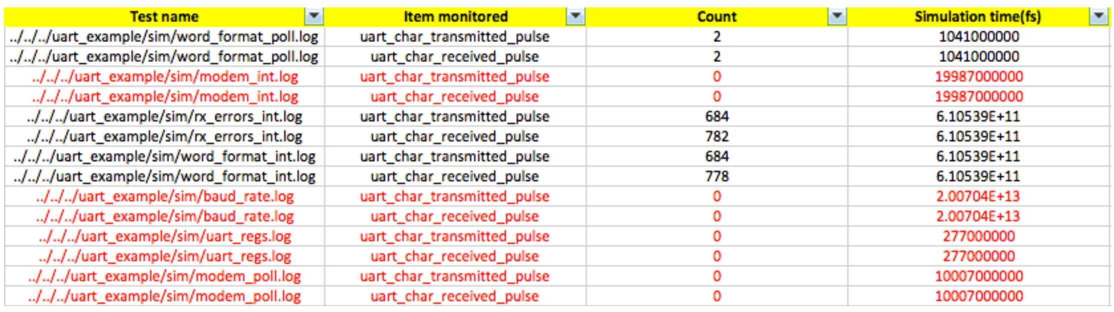
## 2.2 Statistical coverage

Following sections summarize the application of the some of the analytic concepts discussed above using the curiosity tool to the UART test bench.

### 2.2.1 Primary operations

Let's start with something simple.

UART's primary functionality is transmitting and receiving the characters. Every test irrespective of whether it's focusing on feature or configurations verification, it should do some primary operations.



| Test name | Item monitored | Count | Simulation time(fs) |
|---|---|---|---|
| ../../../uart_example/sim/word_format_poll.log | uart_char_transmitted_pulse | 2 | 1041000000 |
| ../../../uart_example/sim/word_format_poll.log | uart_char_received_pulse | 2 | 1041000000 |
| ../../../uart_example/sim/modem_int.log | uart_char_transmitted_pulse | 0 | 19987000000 |
| ../../../uart_example/sim/modem_int.log | uart_char_received_pulse | 0 | 19987000000 |
| ../../../uart_example/sim/rx_errors_int.log | uart_char_transmitted_pulse | 684 | 6.10539E+11 |
| ../../../uart_example/sim/rx_errors_int.log | uart_char_received_pulse | 782 | 6.10539E+11 |
| ../../../uart_example/sim/word_format_int.log | uart_char_transmitted_pulse | 684 | 6.10539E+11 |
| ../../../uart_example/sim/word_format_int.log | uart_char_received_pulse | 778 | 6.10539E+11 |
| ../../../uart_example/sim/baud_rate.log | uart_char_transmitted_pulse | 0 | 2.00704E+13 |
| ../../../uart_example/sim/baud_rate.log | uart_char_received_pulse | 0 | 2.00704E+13 |
| ../../../uart_example/sim/uart_regs.log | uart_char_transmitted_pulse | 0 | 277000000 |
| ../../../uart_example/sim/uart_regs.log | uart_char_received_pulse | 0 | 277000000 |
| ../../../uart_example/sim/modem_poll.log | uart_char_transmitted_pulse | 0 | 10007000000 |
| ../../../uart_example/sim/modem_poll.log | uart_char_received_pulse | 0 | 10007000000 |

*Table: Count of the characters transmitted and received*

If you look at the above table generated by the tool, there are 4/7 tests not transmitting or receiving characters. Tests without primary operations are not really complete tests.

For example, is there a point in changing baud rate (test: baud_rate.log in table) and not doing any traffic? Agreed there is some value in checking if the baud rate generator is working but in real life baud rate functionality is almost useless unless we can transmit and receive characters with changed baud rate. What do you think?

This type of analysis can be very useful where you want to figure out which tests are not doing the primary operations.

### 2.2.2 Tests idling

Another view that confirms the same is percentage of total duration DUT was IDLE in entire regressions. The current UART DUT is IDLE for 94 % of duration of entire regression.

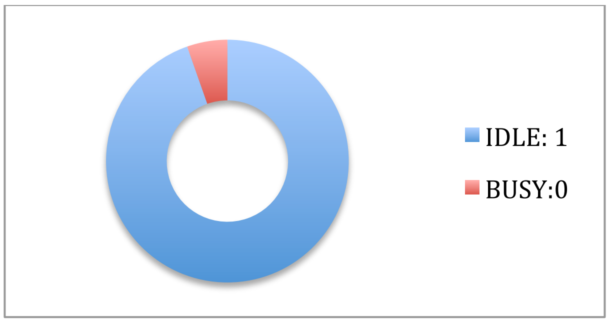


*Figure: Pie char indicating IDLE and BUSY duration in regression*

There is also per test IDLE percentage data generated by analytics that can be used to identify the tests that are idling the most.

This type of insight is very useful for legacy or bulky test benches. Legacy test benches may have hard coded delays. Hard coded delays may mismatch and look bloated when frequency targets are increased in current version compared to last.

### 2.2.3 Configurations

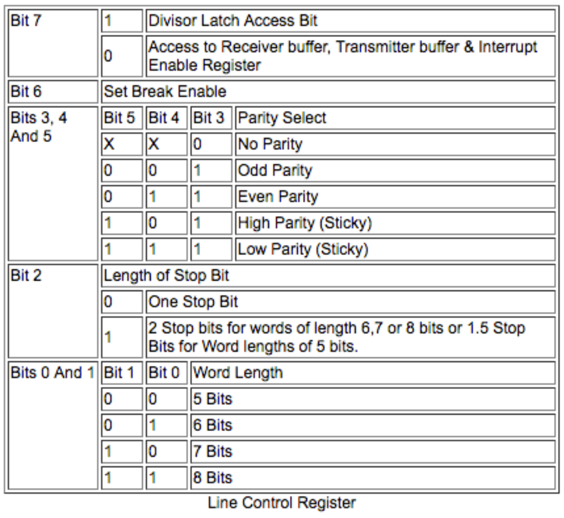
UART supports multiple functional configurations. Let's focus on subset of key protocol configurations. They are number of data bits per character, parity type and number of stop bits.

We would like the regressions to focus on the configurations based on the their relative importance to the application. Depending on application usage, its clear that not all the types of configurations are equally important. So regressions also should reflect the same.

Typically the configurations are randomized at the start of the test and programmed in to the control registers. There on design is exercised with stimulus. In some test cases sub-set of configuration parameters are reprogrammed multiple times during course of test execution.

In order to confirm the regressions are really focusing on the right configurations, we need insights in to per test usage of configurations and regression level configuration usage. Not just listing of unique configurations exercised but listing of it's accumulated active durations as well.

Our configuration parameters of interest are located in LCR registers bits 0 to 5. These 6 bits of configurations lead to 2 ** 6 = 64 unique combinations. Parity configuration bits LCR[5:3] supports 8 combinations but only 5/8 are defined, 3/8 are not defined. RTL defaults to no parity when any of 3/8 combinations are programmed but still functions normally.



| | | | | |
|---|---|---|---|---|
| Bit 7 | 1 | Divisor Latch Access Bit | | |
| | 0 | Access to Receiver buffer, Transmitter buffer & Interrupt Enable Register | | |
| Bit 6 | Set Break Enable | | | |
| Bits 3, 4 And 5 | Bit 5 | Bit 4 | Bit 3 | Parity Select |
| | X | X | 0 | No Parity |
| | 0 | 0 | 1 | Odd Parity |
| | 0 | 1 | 1 | Even Parity |
| | 1 | 0 | 1 | High Parity (Sticky) |
| | 1 | 1 | 1 | Low Parity (Sticky) |
| Bit 2 | Length of Stop Bit | | | |
| | 0 | One Stop Bit | | |
| | 1 | 2 Stop bits for words of length 6,7 or 8 bits or 1.5 Stop Bits for Word lengths of 5 bits. | | |
| Bits 0 And 1 | Bit 1 | Bit 0 | Word Length | |
| | 0 | 0 | 5 Bits | |
| | 0 | 1 | 6 Bits | |
| | 1 | 0 | 7 Bits | |
| | 1 | 1 | 8 Bits | |

Line Control Register

*Table: LCR register definition*

Through Curiosity tool analytics for complete regression, percentage active duration of each configuration is plotted as pie chart below.

Although limited listing is shown on left hand side but all the 64 combinations (<word length>_<stop bit>_<parity type>) have been covered. Obviously a simple a cross coverage would tell us that. But it would not indicate in overall regression or per test level the duration for which each configuration was active.

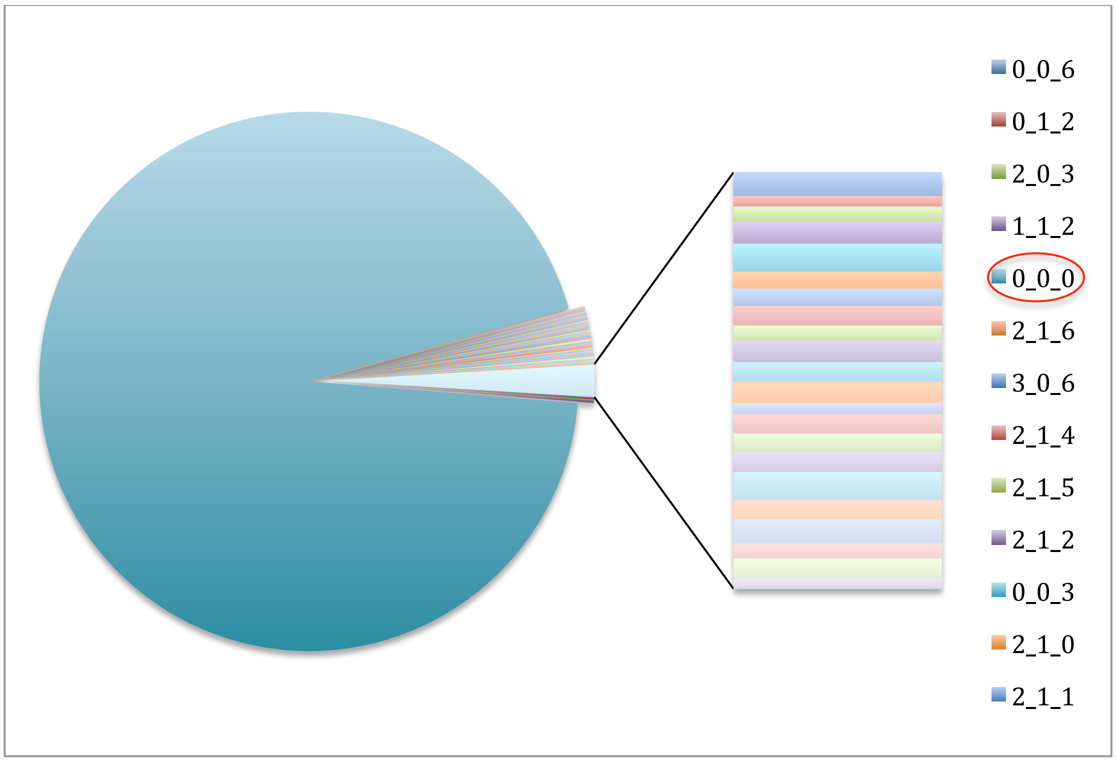


*Figure: Pie chart of percentage active duration of each configuration combination*

From this pie chart, 94.35 % the highest slice of pie is for configuration with all bits of LCR[5:0] being 0 indicated by combination 0_0_0(circled). This maps to configuration: 5-bit per character, no parity and single stop bit.

Now the key question, Is this configuration important enough to be active for 94% of the overall regression time?

Answer may be YES or NO.

It is dependent on the targeted usage. But analytics provides the clarity about where you are investing your valuable simulation time in regression. Time is one the precious commodity, once spent will never come back.

We can also figure out which tests are spending how much time in each configuration? Which tests are dynamically changing configurations? How many times have they changed the configuration?

### 2.2.4 FIFO utilization

This UART design has FIFO in both transmit and receive path. FIFO depth is configured to value 16.

FIFO utilization in overall regression is indicative of nature of traffic and provides some insights to designers about whether the FIFOs are sized optimally.

We looked at the percentage duration for each of the transmit FIFO depth utilization accumulated over all the 7 tests.

Following table on left hand side shows utilization. First column is the number of entries used and second column is percentage of time FIFO stayed at this utilization level.

Just beside that a bar graph plotting utilization count on X-axis and percentage time for that utilization count on Y-axis.

| Utilization | % time |
|---|---|
| 0 | 96.45 |
| 1 | 0.44 |
| 2 | 0.37 |
| 3 | 0.34 |
| 4 | 0.31 |
| 5 | 0.28 |
| 6 | 0.26 |
| 7 | 0.25 |
| 8 | 0.22 |
| 9 | 0.2 |
| 10 | 0.18 |
| 11 | 0.17 |
| 12 | 0.15 |
| 13 | 0.13 |
| 14 | 0.1 |
| 15 | 0.08 |
| 16 | 0.05 |

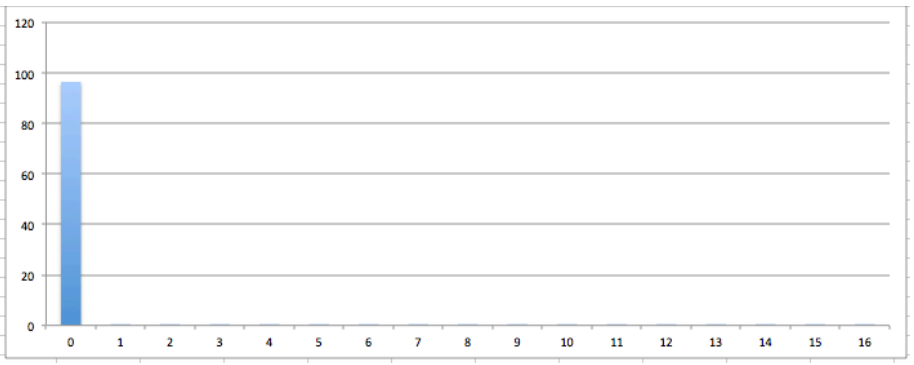


*Table & Figure: Showing the UART transmit side FIFO utilization across regression*

There is one striking observations here. 96 % of simulation time, FIFO is empty. Utilization for different FIFO sizes is so small in comparison to empty that they are not even visible on the bar graph.

If we had added only functional coverage or used the code coverage, it would show up as all the depths being covered since FIFO has become FULL. But if you look at the percentage of overall simulation time the FIFO was full is just about 0.05 %.

### 2.2.5 Interrupt service latency

UART has IRQ signal to request interrupts on various events when enabled. In real life software will have variable latency for servicing the interrupts. Same should be reflected in the test bench as well.

Also some good number of interrupts should be generated and processed. Analytics can provide insights in to this area. Minimum, maximum and average interrupt processing latency and total number of interrupts can be generated to see if they are inline with the expectations.

### 2.2.6 Registers reprogrammed

Control registers are often classified in to one time programmable or dynamically reprogrammable. One time programmable are initialized only once after power on reset and there on they are not touched throughout the operation. While the reprogrammable registers can be reprogrammed dynamically during the operation of device at appropriate states.

Also its not just reprogramming but it should also be reprogrammed with the different value to make it meaningful.

Analytics can provide a quick insight in to how many times each of the registers has been programmed.

### 2.2.7 Some actions based on this exploration:

- Setup an end of test expectation that at least non-zero traffic count is must in all the tests. Curiosity tool can generate such expectations and attach it automatically to all the tests or list of tests. When test fails to meet this expectation error or warning can be flagged
- Tune the constraints on configuration to improve upon the configuration of interest. Reconfirm through analytics that its really taking place. In fact goals can be setup in the curiosity analytics to see if they are being met and can be tracked through the trends
- Traffic patterns need improvement to see better FIFO utilization percentage. If the full case is important then functional coverage on duration of FULL can be setup. Additionally coverage on number of FIFO full and empty cycles can

be created. This type of FIFO utilization cycling can help bring out issues like memory leaks. All this type of functional coverage can be easily generated from the tool using the pre-defined coverage models for the FIFO

## 2.3 Reusable coverage models

Let's look at examples of how reusable coverage models can help us get more insights.

Reusable coverage models are based on best-known verification practices and pattern of typical bugs escaped captured as coverage models. The knowledge base is continuously updated based on new learning's.

Curiosity tool comes bundled with the many built-in coverage models that can be used right out of the box. You can customize, add, share and enforce your organization specific coverage models as well across the teams.

Reusable coverage models are made possible through object oriented functional coverage layering support added to SystemVerilog covergroup construct in the tool.

For serial communication interfaces following are some of the available built-in executable knowledge base. Wherever 'N' is used can be replaced with number that makes sense for your application.

We generated functional coverage for the following items and ran the regressions. Following table summarizes the results.

| Sl. No. | Executable knowledge coverage model type | Coverage results in UART design |
|---|---|---|
| **1** | 16('N') protocol data units transmitted back to back on line without delay<br><br>At least it should be able to transmit one FIFO depth worth of data back | Not covered |
| **2** | 16('N') protocol data units received back to back on line without delay | Not covered |
| **3** | Special patterns being transmitted back to back for 100('N') times<br>• All bits in character 0s<br>• All bits in character 1s<br><br>Serial communication designs have weakness for the data bits being control symbol values | Not covered |
| **4** | Coverage model to check if interrupts are generated when they are masked | **Not generated when masked:** |

| | | |
|---|---|---|
| | Simple interrupt mask implementation mistakes lead to unexpected interrupts. Sometimes in real application these can lead to unintended wake of software leading to additional power consumption. | • rx_int<br>• rx_error<br><br>**Generated when masked:**<br>• ls_int generated when masked<br>• ms_int generated when masked<br>• tx_int generated when masked |
| **5** | Different types of line errors taking place at different level of FIFO utilization | • Break error not injected<br>• Parity and framing errors are injected at various level of FIFO utilization |

*Table: Functional coverage results summary for executable knowledge base items*

## 2.4 Functional coverage amplified

There are many low level routines to help amplify the capabilities of the functional coverage construct.

A bin definition and its reuse for the cover points is one of them. Often the bin definitions are not appropriately parameterized or coded at sufficient granularity. While parameterization affects the portability, lack of granularity affects the quality of the functional coverage.

Imagine a cover point on address hole range with single bin for the entire address range. What values does such cover point provide? So right bin definition is extremely important to extract the full value. These bin definition reuse routines makes this task simpler. Now bins can be reused across cover points either within covergroup or across the covergroup.

| Sl. No. | Simple bin API | Coverage results in UART design |
|---|---|---|
| **1** | Interrupt enable register (IER[3:0]) transitions : get_bins_single_bit_transition_custom() is reused for all the 4-bits of IER register.<br>bins b_transition_1_0_1_0 = (1 => 0 => 1 => 0); | Not covered<br><br>None of the bits, transition is seen |
| **2** | Slave address range decode error. Current APB address bus supports 32 byte address. 8/32 are used. 9-31 byte address is hole. Need to cover both read and write access to this hole generates slave error. | Not covered<br><br>Neither Read or Write issued to byte address hole |

| | | |
|---|---|---|
| | get_min_max_intermediate_equal_ranges_bin_list(9, 31, 3)<br>Automatically generates the following bin ranges: Note that start(9) and end(31) corner cases are covered with bin.<br>bins bin_value_9 = {9};<br>bins bin_value_31 = {31};<br>bins bin_range_10_16 = {[10:16]};<br>bins bin_range_17_23 = {[17:23]};<br>bins bin_range_24_30 = {[24:30]}; | 9-31 while valid address 0-8 is accessed multiple times |

*Table: Functional coverage summary for items built using bin definition reuse*

## 3 Conclusion

Stimulus quality analysis tool like curiosity enables you to get quick insights and giving instantaneous feedbacks about your functional verification quality.