Automated Generation of Functional Coverage Metrics for Input Stimulus

by Mike Andrews, Verification Technologist, Mentor Graphics

Questa® inFact intelligent testbench automation has allowed many verification teams to achieve their initial coverage goals far more efficiently than would have been possible with traditional constrained random generation, providing an opportunity to expand those coverage goals to ensure a more comprehensive verification of the DUT. This shifts the verification engineer's challenge from efficiently achieving coverage to efficiently and accurately defining the additional covergroups and coverpoints required. This process can be especially difficult when defining cross coverage goals where there are multiple constraints that limit the legal combinations of the variables concerned. Determining the correct expressions required to exclude the illegal combinations from the cross is both time consuming and error prone. Questa inFact allows for graphical definition of the coverage goals and can, with the 10.1 release, automatically generate SystemVerilog covergroups from this definition, including the exclusions needed to accurately represent the achievable coverage. This article describes how this capability can simplify the definition of more comprehensive stimulus coverage metrics.

INTRODUCTION

Verification teams are always under pressure to meet their project schedules, while at the same time the consequences of not adequately verifying the design can be severe. This puts the team between a rock and a hard place as they say. The main value of Questa inFact is to help with the problem of meeting the schedule requirements by more efficiently, and more predictably, generating the tests needed to meet coverage goals in the case where coverage metrics are being used to determine 'completeness' of the verification project. An indirect benefit of this has always been that, when planning to use this capability on a project, a verification team can expand the scope of the functional coverage metrics and therefore can expect to avoid the prior mentioned severe consequences of letting serious bugs slip through. This has however appeared to move the bottleneck in the process to the creation of these expanded coverage metrics. If many additional cross coverage goals are added this can be especially time consuming, and actually many bugs can be, and have been in my experience, introduced into the coverage scoreboard during this process. This issue has definitely slowed down the overall move to true coverage driven verification processes in some organizations, forcing them to fall back on easier, but less reliable, metrics such as code coverage.

DEFINING CROSS COVERAGE IN THE PRESENCE OF COMPLEX CONSTRAINTS

The real difficulty in defining cross coverage goals is to get the exclusions correct, and this can be really difficult when the variables in the cross have many complex constraints that determine the relationships that must be maintained between them. If these exclusions are not properly identified then the ability to accurately gauge the actual coverage achieved becomes impossible since the missing coverage may contain a significant percentage of illegal cases. Or, quite often, the exclusions defined are incorrect and important cross coverage combinations are not tracked by the metrics, further obfuscating the actual coverage as reported by the verification metrics.

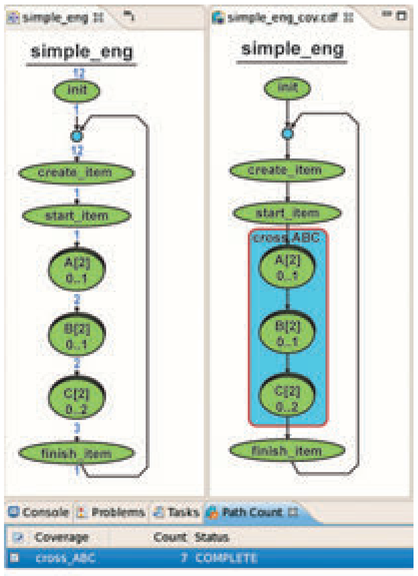
To illustrate the difficulty let's consider a very simple three variable example:

```
class my_item extends uvm_sequence_item

rand bit A;
rand bit B;
rand bit [1:0] C;

constraint limC { C < 3; }

constraint relate_vars {
        if (A == 0) {
                        B == 0;
                        C == 0;
                        };
        }

endclass: my_item
```

Our coverage goal is to test all of the legal combinations of all three variables A, B and C. The definition of the legal cross coverage goals is not trivial even for this case, since we have to determine which combinations of A,B,C are unreachable. This can take a little thought to determine by hand. The illegal combinations turn out to be:

```
A[0], B[1], C[0,1,2]
A[0], B[0], C[1,2]
```

This leaves 7 valid combinations out of a total of 12, which would mean that if the exclusions are not specified then the maximum coverage attainable for the cross is about 58%. This is actually a fairly common percentage for most real testbenches that I have encountered, and verification engineers often struggle to determine if this result is actually a good level of coverage or whether the missing coverage contains a significant portion of important legal variable combinations. Questa inFact has been able to help with this problem for quite some time since it can provide an accurate count of the legal solutions in a cross, as shown in Figure 1 below.



Coverage	Count	Status
cross_ABC	7	COMPLETE

Figure 1. Accurate Counting for Cross Coverage Goals

In Figure 1 we have three views of the same stimulus item variables. The first (top left) shows the full state space with a count of the total number of combinations as described in the stimulus graph. The second view (top right) shows an annotation of a cross coverage goal on the graph. The third view (bottom) is a count of the legal combinations as determined from a combination of the variable types and the constraints placed on the combinations of those variables. From this an engineer can determine at least the actual achievable coverage for a particular cross. There is still a potential issue however if there is an error in the definition of the exclusions so that we end up with a similar count in the metrics, but we have inadvertently flipped a bit somewhere in our exclusion logic. If this is the case then we still end up with a mismatch in the coverage metrics score reported vs. the real situation, and the engineer still has to try to determine where the problem lies, i.e. in the stimulus generation process or the covergroup itself.

AUTOMATED GENERATION OF SYSTEMVERILOG COVERGROUPS

A recent enhancement to Questa inFact brings significant additional value for this kind of problem. The user now has the option to create a SystemVerilog covergroup that is based on the inFact coverage strategy defined in the tool's IDE. The covergroup so created will include automation of the necessary exclusions based on the constraints defined on the variables in the stimulus class. Following is the actual coverage code as created by inFact for this simple case.

```
// Path Coverage cross_ABC
  cross_ABC_A_cp : coverpoint m_cov_item.A {
     option.weight = 2;
     bins A[] = {0, 1};
  }
  cross_ABC_B_cp : coverpoint m_cov_item.B {
     option.weight = 2;
     bins B[] = {0, 1};
```

```
        }
        cross_ABC_C_cp : coverpoint m_cov_item.C {
           option.weight = 3;
           bins C[] = {0, 1, 2};
        }
        cross_ABC : cross cross_ABC_A_cp, cross_
ABC_B_cp, cross_ABC_C_cp {
           option.weight = 7;
           ignore_bins unreachable_bins =              (
                 (binsof(cross_ABC_A_cp) intersect {0} &&
 binsof(cross_ABC_B_cp) intersect {0} && binsof(cross_
ABC_C_cp) intersect {1,2}) ||
                 (binsof(cross_ABC_A_cp) intersect {0} &&
binsof(cross_ABC_B_cp) intersect {1} && binsof(cross_
ABC_C_cp) intersect {0,1,2})
            )
;
        }
```

Figure 2. inFact Generated Covergroup Code

In Figure 2, note the ignore_bins statement in the cross coverage definition. Having written some of these types of statements for more complex situations myself, I definitely appreciate the apprehension with which a verification engineer might have approached that task without tools to help. This fear may certainly have been weighed against the promise of taking advantage of the power of inFact 's algorithms to confidently define more, and larger cross coverage goals in their functional coverage metrics.

SAVING TIME IN FUNCTIONAL COVERAGE CREATION

This new capability in Questa inFact opens up the possibility of an improved process for creating functional coverage metrics, at least for the covergroups that track stimulus coverage for the sequence_items in a testbench.

Once the sequence_item itself has been created, another facility of inFact can be used to import the information from the sequence_item, including the definition of the randomize-able variables, and the constraints placed upon them. From this imported information a graph is automatically created to generate valid values for that item.

A coverage strategy can then be determined based on the graph variables, using the tools in the inFact IDE to get an accurate picture of the number of legal combinations in the cross coverage goals being defined.

Figure 3 on the opposite page shows a more complex sequence_item example which is part of a testbench for an ethernet controller. This example shows a clearer picture of how these tools can be used.

Figure 4 on page 19 shows an inFact coverage strategy defined for a testbench generating ethernet traffic for a controller DUT.

In Figure 4, there are two cross coverage goals defined, each of which targets a different subset of the graph variables. For each cross coverage goal in the strategy, the colored region defines the variables to be considered, with some variables in the region defined as 'Don't Care' as denoted by a white oval around the variable. For visual clarity, each of the goals can be viewed separately if required, which is important if there are many different combinations.

The specific bins for each of the cross coverage goals have been iteratively defined, using the path counting feature (shown at the bottom of the figure) to roughly balance the sizes of the goals. The largest of these goals gives a general idea of the number of items that need to be generated and simulated to meet the coverage goals for this particular item (although, depending on the constraints between all the variables, meeting all the goals may take a few more items than the 288 shown in Figure 4).

From the coverage strategy defined in inFact, a covergroup can now be automatically created that will track the actual coverage metrics as determined by the SystemVerilog coverage features of the user's simulation environment. All this without the need to painstakingly analyze the constraints to manually determine the exclusions needed for accurate tracking of the coverage progress.

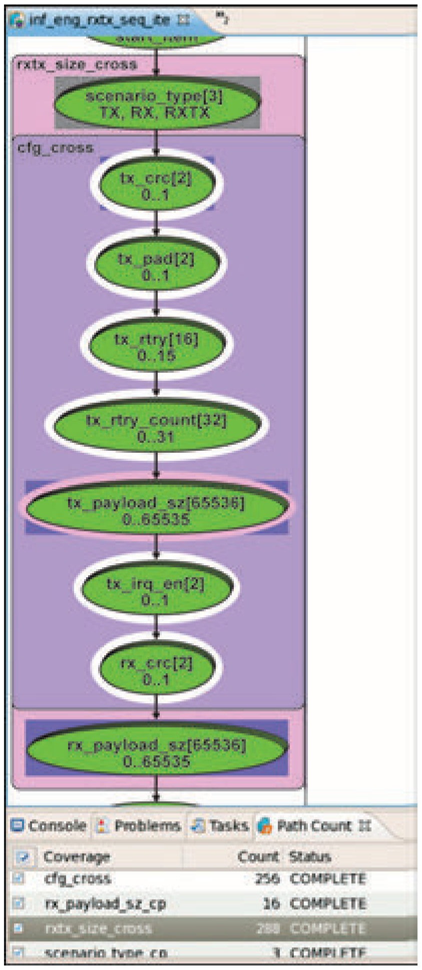
To get a sense of the potential time saved, take a look at the 'eye-chart' in Figure 5 on page 18, which is the actual generated coverage code that defines one of the crosses for this example, with the ignore_bins statements required to properly exclude illegal combinations.

```
typedef enum {
        TX,
        RX,
        RXTX
} ethmac_rxtx_scenario_t;

class ethmac_rxtx_seq_item extends uvm_sequence_item;

        `uvm_object_utils(ethmac_rxtx_seq_item)

        rand ethmac_rxtx_scenario_t         scenario_type;
        // Enables
        rand bit                                                tx_crc;
        rand bit                                                tx_pad;
        // Control the retry limit to program, and the
        // number of retries the MII driver must provoke
        // Retry limit to be programmed into the BD
        rand bit[3:0]                                           tx_rtry;
        // Retry count for the MII driver
        rand bit[4:0]                                           tx_rtry_count;
        rand bit[15:0]                                          tx_payload_sz;
        rand bit                                                tx_irq_en;
        // Provoke a CRC error
        rand bit                                                rx_crc;
        rand bit[15:0]                                          rx_payload_sz;
        rand bit                                                rx_irq_en;

        // Constraints to zero some fields based on scenario type

        constraint scenario_fix_c {
                if (scenario_type == RX) {
                        tx_crc  == 0;
                        tx_pad  == 0;
                        tx_rtry == 0;
                        tx_rtry_count == 0;
                        tx_payload_sz == 4;
                        tx_irq_en == 0;
                } else if (scenario_type == TX) {
                        rx_crc == 0;
                        rx_payload_sz == 4;
                        rx_irq_en == 0;
                }
        }

        constraint bug_c {
                // Appear to be issues with enabling pad. TX State Machine hangs.
                tx_pad == 0;
        }

        constraint valid_c {
                tx_payload_sz inside {[4:8192]};
                rx_payload_sz inside {[4:8192]};
        }

endclass
```

Figure 3. Sequence Item Example for Ethernet Testbench

```
        cfg_cross : cross cfg_cross_tx_crc_cp, cfg_cross_tx_pad_cp, cfg_cross_tx_rtry_cp, cfg_cross_tx_rtry_count_cp,
cfg_cross_tx_irq_en_cp, cfg_cross_rx_crc_cp {
            option.weight = 256;
            ignore_bins unreachable_bins =                (
                (binsof(cfg_cross_tx_crc_cp) intersect {0} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {0} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {0} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]}) ||
                (binsof(cfg_cross_tx_crc_cp) intersect {0} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {0} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {1} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]}) ||
                (binsof(cfg_cross_tx_crc_cp) intersect {0} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {1} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {0} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]}) ||
                (binsof(cfg_cross_tx_crc_cp) intersect {0} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {1} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {1} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]}) ||
                (binsof(cfg_cross_tx_crc_cp) intersect {1} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {0} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {0} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]}) ||
                (binsof(cfg_cross_tx_crc_cp) intersect {1} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {0} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {1} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]}) ||
                (binsof(cfg_cross_tx_crc_cp) intersect {1} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {1} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {0} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]}) ||
                (binsof(cfg_cross_tx_crc_cp) intersect {1} && binsof(cfg_cross_tx_pad_cp) intersect {1} && binsof(cfg_
cross_tx_irq_en_cp) intersect {1} &&
                    binsof(cfg_cross_rx_crc_cp) intersect {1} && binsof(cfg_cross_tx_rtry_cp) intersect
{[0:3],[4:7],[8:11],[12:15]} &&
                    binsof(cfg_cross_tx_rtry_count_cp) intersect {[0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]})
              )
;
        }
```

Figure 5. Cross Coverage Goal Definition For Ethernet Config



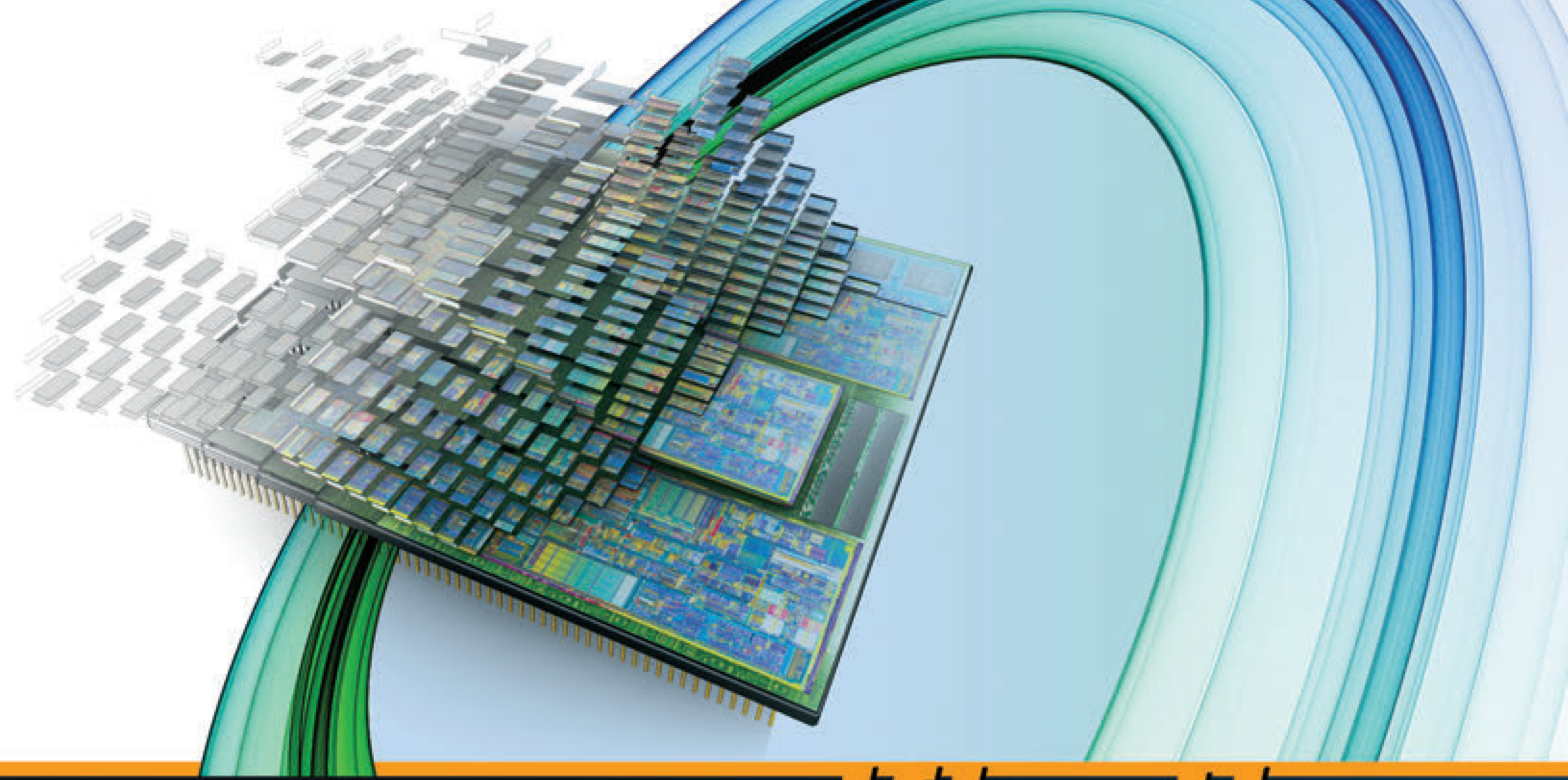
Coverage	Count	Status
cfg_cross	256	COMPLETE
rx_payload_sz_cp	16	COMPLETE
rxtx_size_cross	288	COMPLETE
scenario_type_cp	3	COMPLETE

Figure 4. Coverage Strategy for Ethernet Rx/TX Sequence Item

What would normally take a number of hours to define, and subsequently debug, can now be automatically generated, leaving time to focus on other aspects of coverage such as embedding assertions or more critical analysis of the overall verification goals.

CONCLUSION

If meeting coverage goals is one of the current pain points in a verification project, then intelligent testbench automation solutions such as Questa inFact can significantly help address this. This latest capability — the automated generation of coverage code — is intended to proactively attack what may be the next bottleneck, i.e. the efficient and accurate definition of more comprehensive coverage goals that the intelligent automation can then target (with equal efficiency).



verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Mentor Graphics®

www.mentor.com