Who's Watching the Watchmen?

The Time has Come to Objectively Measure the Quality of Your Verification Environment

David Brownell
Analog Devices, Inc.
Norwood MA
david.brownell@analog.com

Abstract

With the adoption of SystemVerilog and UVM the size and complexity of testbench code has increased dramatically over the years, to the point where the testbench code often dwarfs the size of the RTL design code to be verified. Along the way it was recognized that bugs in the testbench code are as dangerous as bugs in the RTL and that a method to objectively measure the quality of the entire verification environment was required.

The traditional methods of Code and Functional Coverage were a good start, providing insight into the stimulus capabilities of testbenches, but these metrics provide no visibility into whether the scripts, tests, and checkers that make up the whole verification environment are capable of detecting bugs if they exist in the RTL. Functional Qualification tools are intended to fill this gap. This paper will describe how we have successfully used one such tool, Synopsys/Springsoft's Certitude, to fill this void in our verification signoff process. Topics covered will include integrating Functional Qualification into different verification environments, running the tool, analyzing results, and examples of real issues found in our verification environments.

Keywords—Functional Qualification, Certitude, Functional Verification, Verification Quality

I. INTRODUCTION

Verification is hard. Whether the verification environment consists of a simple Verilog testbench with directed test cases or a full blown System Verilog constrained-random coverage-driven environment based on the latest UVM release, verification engineers all struggle with the same questions. Are my tests exercising all the important scenarios? Am I checking everything? What did I forget? Am I done? In the end, effective verification always comes down to same three words: Activation, Propagation, and Detection. For every potential bug in the design the verification environment must have at least one test capable of activating the bug, propagating the result of the bug to a checker, and finally the checker must report a failure. An oversight in any one aspect means potential bugs remain in the design.

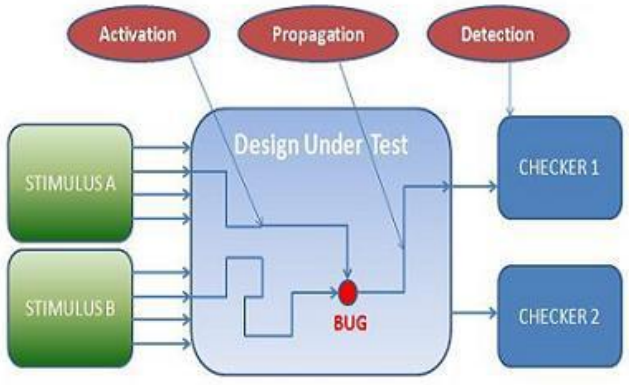

Figure 1. Requirement for Effective Verification

The traditional method to determine if your verification effort was effective and to answer the questions posed earlier has been to collect and analyze code and functional coverage. There is an inherent problem with relying solely on these metrics; while they do an excellent job of measuring the activation capabilities of a verification environment, they provide no insight into the propagation or detection capabilities of the testbench. These coverage metrics rely on the unfounded belief that if the environment is capable of activating a bug then it is also capable of propagating and detecting that bug.

Given the size and complexity of modern testbenches and the fact that coding mistakes are just as likely in the testbench as the design this is a very risky assumption to make. In order to truly measure the effectiveness of an environment one must be able to measure that environments ability to activate, propagate and detect bugs within a design. This is exactly what Functional Qualification tools are able to do.

II. FUNCTIONAL QUALIFICATION

A. *A Brief History of Functional Qualification*

Functional Qualification for design verification was inspired by mutation analysis in the software development domain where research has been ongoing since the 1970s. The foundation of mutation analysis is the idea that one can identify weaknesses in the testing of a software program by systematically introducing small behavioral changes, called "mutations", into the program and checking that the mutation is observed by the validation suite. If there is a difference in the output of a test between the mutated code and the original code then the mutation is considered "killed". However, if no difference is observed then the mutation is considered "live" and indicates a potential weakness in the testing of the software program. The next principal of mutation analysis is that by improving the testing of the program to kill the mutants one can potentially identify real bugs that are not capable of being detected in the current environment. Table 1 below shows example mutations that could be inserted into a program.

TABLE I. MUTATION EXAMPLES

Original Code	Mutated Code
A = B + C;	A = B - C;
If (A || B)	If (A && B)

The major limitation of mutation analysis is that it can be extremely computationally expensive; all mutations are considered in isolation and the entire test suite must be run for each mutation to determine if it is killed or live. In large software systems the amount of time required to consider all possible mutations can exceed available resources.

In today's Functional Qualification software for semiconductor verification the term "mutation" has been replaced with "fault", but the underlying principle is the same. The software systematically introduces faults into the design code and runs the verification test suite to see if the faults are detected by the testbench. A fault is considered detected if a passing test fails when the fault is in place, and any fault that is not detected highlights a potential weakness in the verification environment. Fixing these weaknesses by adding new tests or improving the testbench checkers, increases the chances of exposing any real bugs that remain in the design. Today's software also employs algorithms for test ordering, duplicate fault analysis, and fault dropping to improve the computational efficiency of the analysis.

B. *Faults*

A common argument raised against functional qualification is that it is not possible to model every coding mistake that users will make, and therefore there can still be bugs in your design after completing the effort. Both of these statements are true, however, they are also not claims made by users of functional qualification tools.

It is important to remember that the faults introduced during functional qualification are not intended to replicate bugs that a user might make in the code; they are only intended to introduce a change of functionality into the design. If the verification suite is not capable of detecting this change, the claim of functional qualification users is that there is a potential weakness in the verification environment. Furthermore, improving the testbench to detect this change the likelihood will increase of exposing real bugs that may still exist in the design.

Since the purpose of functional qualification is to find weaknesses in the verification environment and that this process can be very computationally expensive; faults are organized into classes based on how large of an impact they are likely to have on the design. A fault in a "higher" class is more likely to fundamentally change design behavior when compared to a fault in a "lower" class, and therefore higher class faults should be easier to detect by the verification environment. The higher the class of a non-detected fault, the higher the likelihood of a major weakness in the verification environment.

For example, a stuck-at fault placed on a module output is much more likely to create a detectable change in behavior during a test run, than a fault changing a conditional operator from "||" to "&&" inside of the same module. If the testbench cannot detect a stuck-at fault on a module output, there is potentially a major weakness in the environment checkers, and it is unlikely that the testbench would be capable of detecting a more subtle conditional change inside module. Higher class faults are submitted for analysis, before lower class faults and non-detectable faults in higher classes should always be analyzed before lower class faults. Below are 4 classes of faults we typically consider during functional qualification, from highest to lowest.

1) ***Connectivity Faults*** *:* This class of faults consists of stuck-at-0, stuck-at-1, or negation faults on the module inputs and outputs throughout the design hierarchy. These faults tend to enable/disable functionality throughout the design and should be easily detectable by a verification environment. A non-detected connectivity fault can indicate a major issue in the verification environment such as completely missing checkers or tests. All non-detectable connectivity faults must be resolved before proceeding with functional qualification of lower class faults, as these faults gate the propagation of faults inside the module and it will be a waste of resources to continue.

2) ***Reset Faults*** *:* The reset fault class consists of a single fault targetting a common weakness found in verification environments related to reset. Often, reset is asserted once at the beginning of tests and held steady for the remaining duration of the tests. Therefore, none of the conditional logic associated with reset is fully exercised or verified. The fault in

this class forces the first conditional statement inside an always @(reset) block to evaluate to true, which will quickly highlight if the verification environment is fully checking the reset states of the DUT.

3) ***Synchronous Control Faults*** *:* There are a large number of specific fault types inside the Synchronous Control fault class that are intended to change functionality of control statements such as if/then/else, or case within always blocks inside the design. Examples of these faults include faults forcing conditional statements to evaluate to always true, always false, or negate the entire conditional statement. These faults can also completely remove cases items from case statement trees or else conditions from if statements, forcing the alternate paths to be chosen. These faults tend to remove a small piece of functionality from the design or slightly alter the path taken by tests. If a verification environment cannot detect a fault in this class usually indicates a weakness with an existing checker or missing testcase.

4) ***Logic Faults*** : The remaining fault types are generally classified as logic faults that can exist inside and outside of synchronous control code. This a rather large fault class that includes conditional faults forcing conditions to evaluate to always true or always false as well as operator faults to change various operators such as "or" to "and" or "+" to "–". The last group of faults in this class is directed towards busses defined in the design and includes faults which force stuck at faults on the first and last bits of the bus as well as faults to negate the entire bus. Non-detected faults in this class tend to be difficult to diagnose and resolve as they are often associated with corner cases in the design. Ideally all of these faults would be resolved with additional checkers and tests, but often this is not the possible due to schedule and resource constraints. Because of the large amount of time required to analyze and kill these faults. This group should only be considered when all faults in the higher classes are detected.

C. The Functional Qualification Process

Functional qualification of a verification environment consists of three distinct phase: the model phase, the activate phase, and the detect phase.

1) ***The Model Phase****:* In this first phase users configure the RTL files to be targeted for qualification as well as the types of faults that are allowed to be inserted. The tool then parses the files to determine all of the possible locations that faults can be inserted and creates instrumented versions of the code for use in the following two phases.

Table II shows how two example faults could be instrumented into design code. One fault completely removes an assign statement so the target variable never gets updated, and the second changes an "or" operator to "and" changing the output of the assign. The selection of which fault is active during simulation can be controlled with plusargs passed through the simulation command or using the simulator PLI interface.

TABLE II. FAULT INSTRUMENTATION

Original Code	Instrumented Code

Always @(posedge clk)
A <= (B || C);
	Always @(posedge clk)
	
	//remove assign fault
	If (FAULT_ID == 1)
	begin end
	
	//or operator replace with and
	else if (FAULT_ID == 2)
	A <= (B && C) ;
	
	// original code no faults
	else
	A <= (B || C) ;

In an effort to minimize the time required to complete the next two phases, the functional qualification software performs analysis to eliminate equivalent and unreachable faults and determines cones of influence for each fault during the model phase. The cone of influence information is used in the final phase so that when a fault is not detected, all the related faults can be dropped from the analysis while the initial fault is evaluated.

2) ***The Activate Phase****:* The first step in the Activate phase is for the user to configure all the tests to be used in the qualification and if the testbench is a constrained-random environment the user must also configure which seeds to use for each test. The software will then run each test one time to determine the runtime and which faults are "Activated" by each test. A fault is considered "Activated" by a test if the line of design code containing the fault was executed at least once during the test.

If no test in the entire test suite executes the lines containing the fault then that fault is considered "Non-Activated". These faults will often match up with missing line coverage in the design's code coverage report "Non-Activated" faults can mean there are tests missing in the verification suite, tests are over-constrained, or possibly that there is dead code in the design. There are also valid reasons for these lines not be activated and just as in code coverage where users can exclude faults, users can easily disable faults from consideration in the functional qualification tool as well.

At the end of this phase the it will be known for every fault, that either the fault is "Non-Activated" or that the fault is at least capable of being detected and exactly which tests are capable of doing so. This information allows the runtime in the detect phase to be kept to the absolute minimum, by ordering the tests to run from the shortest to the longest in duration.

3) ***The Detect Phase****:* In this final phase each fault is considered in isolation by inserting the fault and running the tests that activate the fault serially, from shortest to longest, to determine if any test detects the fault. A fault is considered detected by a test if the test fails when the fault is inserted and passes when it is not present. In functional qualification terms a fault is considered "Detected" if any one test from the full test suite detects the fault.

If no test which activated a fault fails when the fault is inserted then there are two classifications that fault will fit into. The first is "Non-Propogated", which means there was no difference observed at the "output of the design" when compared to the same test with no fault inserted. The second is "Non-Detected" which means that there was a difference in the "output of the design" compared to the no fault test, but the test checkers did not report a failure. "Non-Detected" faults are a strong indication of weaknesses or bugs in the verification environment.

"Output of the design" typically refers to the top level output pins of the design under test, where verification checkers are typically located. However, any node in the design hierarchy can be declared as "output of the design", and the only difference between a "Non-Propagated" and a "Non-Detected" classification is this definition.

Both classifications indicate probable weaknesses in the verification environment and should be analyzed further to understand why the verification environment is not capable of detecting these faults. We have observed that "Non-Propagated" faults often indicate missing tests in the evaluation; whereas "Non-Detected" faults indicate weaknesses within current checkers or that no check exists for some outputs of the DUT.

D. ***Results of Functional Qualification***

At the end of a functional qualification effort every fault will have been categorized as shown in table III, and users will know for every line of RTL if the verification environment is at least capable of finding a bug if it exists on the line.

TABLE III. FAULT CLASSIFICATION

Category	Description
Non-Activated	No test capable of activating the fault
Non-Propagated	Fault Activated, but not propagated to a checker
Non-Detected	Fault propagated to checker, but no fail reported
Detected	At least one test reported a failure

Throughout the remained of this paper the following abbreviations will be used for fault classifications: Non-Activated (NA), Non-Propagated (NP), and Non-Detected (ND).

These classifications are the distinct advantage of functional qualification compared to code coverage and functional coverage metrics. With these traditional verification metrics, when an item is "covered" all one knows is that a test stimulated a particular line of code or defined coverpoint. There is no information about whether the testbench is observing and correctly checking the result of executing the code being covered. The different fault classifications of functional qualification provide all of this information.

Another benefit of functional qualification is the information gathered on the tests in your verification suite. For every test you will know the simulation time, number of faults activated by the test, count of faults propagated by the test, and total number of faults detected by the test. With this information the verification team can optimize regression test ordering based on test time and number detected faults, and eliminate tests that are not providing any value.

As with any verification metric there has to be a minimum criteria to determine when the effort is complete. Within our organization the signoff criteria for verification complete status is zero NA and zero ND faults, after waivers. For NP Faults the current target is less than 10% of the total faults, but this varies depending on the project.

One of the biggest surprises found when we implemented functional qualification was the number of NP faults in our environments. These are faults that would typically be reported as covered with our old metrics but in reality were never being checked. Reducing the number of NP faults can be as simple as adding more seeds for constrained-random tests in the qualification or as time consuming as adding more checkers and assertions throughout the design hierarchy to increase the odds of observing faults.

III. OUR EXPERIENCES WITH FUNCTIONAL QUALIFICATION

A. ***Integrating Functional Qualification Software***

Over the last year and half we have integrated functional qualification into the verification process for more than 10 separate projects in three different organizations within the company. The environments analyzed have ranged from small block level testbenches to full system level SOC verification testbenches, and in all cases the integration has been very straight forward.

On average we have seen it takes new users 1 to 2 days to get functional qualification software running on their initial project, but for follow-on projects the setup time is typically less than half of a day. The process of preparing an environment for functional qualification can be automated if there is a high level of standardization in your environments

and this has been done by some groups reducing the setup time to a few minutes.

Table IV below shows the size of the design and testbench for a few of the environments we have analyzed. Most of these environments are written in SystemVerilog, constrained-random focused, and based on either VMM or UVM. Notice that the testbenches are generally larger than the designs they are verifying, with the one exception being the small SOC testbench. This was the only testbench written in standard Verilog using primarily directed tests and interestingly, it is also the environment with the highest percentage of non-detected faults!

TABLE IV. ENVIRONMENTS QUALIFIED

Testbench	RTL Line Count	TB Line Count
Peripheral Block	~9000	~14000
Processor Sub-block	~15000	~22000
Memory Controller	~18000	~42000
Small SOC System	~90000	~17000
SOC System	XXX	>250000

Setting up testbenches for evaluation consists of creating configuration files to tell the functional qualification software five things: what files to analyze, what fault types insert, what tests to use, how to compile the testbench, and how to run a test and determine if it passed or failed.

The most critical, and sometimes overly complicated task, is defining for the functional qualification software what constitutes a pass or fail when a test is run. A mistake in this configuration file can easily lead to bad results where non-detected faults are falsely reported as detected. To ensure valid results the verification environment should employ a standard messaging system which reports at the end of every test that the simulation ran to completion and the number of failures. Allowing test creators to print their own error messages and relying on complicated post-processing scripts to track errors can lead to false passes in your regression runs and complicates qualification runs. The good news is that if your testbench currently uses this method then functional qualification will highlight any holes you do have.

*B. **Running the Analysis***

*1) **The Model Phase*** : As described earlier the model phase of qualification is where the targeted RTL files are instrumented with the faults to be inserted during the analysis. This phase is typically the shortest of the three phases and in our experience takes between 1 minute and 1 hour, depending on the size and complexity of RTL files being targeted for qualification. Table V. shows the number of faults instrumented and the corresponding model phase run times for the environments shown earlier. The SOC system TB is not shown as only a subset of the system was targetted for analysis.

TABLE V. MODEL PHASE RESULTS

Testbench	Faults Inserted	Runtime
Peripheral Block TB	5789	2 min
Processor Sub-block TB	7512	3 min
Memory Controller TB	8476	5 min
Small SOC System	44812	46 min

*2) **The Activate Phase*** : The activate phase is where each test is run exactly one time to determine which faults are potentially detectable by that test. The runtime for this phase can vary greatly as it is directly determined by the number and length of the tests included for analysis.

We have found that the first time you run functional qualification on an environment you will invariably identify non-detectable faults and for this reason we recommend limiting the initial test suite to a set of tests that can complete this phase in less than 1/2 hour. This is because many faults are activated by nearly every test, and if they are not initially detecable every test will be run for each these faults, quickly leading to excessivly long run times in the detect phase.

Keeping the initial runtime short ensures a fast turnaround on the first pass of the detect phase which can help to quickly identify major holes in the environment. Once the initial pass has been completed and early non-detects resolved one can add the full test suite and rerun the activate and detect stages to complete the full analysis.

The primary purpose of the the activate phase is to identify the non-activated faults in the design which are not exercised by any test in the evaluation suite. Analyzing these faults to determine if they are located within dead code and can be diasabled or within valid code and additional tests are required is identical to the process of debugging and excluding code coverage. If you already have code coverage exclude files for the design you can easily convert them to work with the functional qualification software.

TABLE VI. ACTIVATE PHASE RESULTS

Testbench	NA Faults Found	Runtime
Peripheral Block TB	49	12 min
Processor Sub-block TB	461	9 min
Memory Controller TB	146	10 min
Small SOC System	3277	39 min

3) ***The Detect Phase :*** This final phase of analysis is the most time consuming, both in terms of compute resources to analyze the propagation and detection of faults, and in engineering resources to debug the NP faults and ND faults that are found. An environment with a high number of NP and ND faults will take longer to run than the same environment that is capable of detecting all the faults. This is because faults must run through all tests that activate the fault to be considered as NP or ND, whereas soon as a single test is found that is capable of detecting a fault, analysis is complete for that fault. When NP or ND faults are identified the software will "drop" faults in the cone of influence for these faults, to speed up the analysis. This means once the NP and ND faults are resolved analysis will have to be run again to evaulate the dropped faults.

We have observed that it can take weeks for large designs to complete a single qualification run and have found it is best to take an iterative approach. We now run the analysis until five ND faults are identified and then debug and resolve these faults before continuing. The tables VII and VIII below show some the initial and full runtimes for various environments.

TABLE VII. INITIAL DETECT PHASE RESULTS

Testbench	ND	NP	Detected	Not Qualified	Runtime
Peripheral	5	62	208	5514	62 min
Proc. Sub-block	5	340	86	7081	5 min
Mem Controller	5	88	267	8116	100 min
SOC System	5	1432	23	43352	30 min

TABLE VIII. FULL DETECT PHASE RESULTS

Testbench	ND	NP	Detected	Not Qualified	Runtime
Peripheral	30	74	762	4923	8hr 10m
Proc. Sub-block	16	411	5258	1827	7hr 42m
Mem Controller	23	102	299	8052	6hr 21m
SOC System	309	1496	1006	42001	>7 days

When ND and NP faults are identified they must be analyzed to determine why the testbench did not detect them. To do this we always go through the following steps. First, we identify the type of fault and where it was inserted in the design. Then we can compare the original test waveforms to the waveform with the fault and work with the designer to understand the change in functionality caused by the fault. Once the change is understood we compare the behavior to the design specification and test plan to see if the testbench should identify this change as a bug. If the answer is no then we disable the fault, if the answer is yes then a new test or checker must be added to the environment and the analysis is run again for this fault.

C. Our Results: Every Environment has Problems

When we first began implementing functional qualification as part of our verification signoff process we expected to find few if any ND and NP faults as we were already using 100% code and functional coverage for signoff metrics. We quickly learned that no environment is perfect and in fact all of our environments had weaknesses and many had significant holes. Every team that has adopted functional qualification in our organization has had similar results where every testbench has problems; the following paragraphs will highlight the most common weaknesses found behind ND and NP in all of the environments that have been analyzed.

1) ***Missing Checkers on DUT outputs:*** The most basic principal of proper functional verification is that all output pins must be monitored and checked for correctness. We have found when functionally qualifying our verification enviroments that in nearly every case pins that we thought were being monitored were in fact not being checked at all.

Most often these are control pins as part of an interface where the verification engineer simply forgot to put a check in place, and because the pins are toggling during tests functional coverage and code coverage report the pins as covered so the mistake goes unnoticed. In one case we found missing checks on a the lower portion of a data bus where an engineer mis-understood the protocol and incorrectly removed the checks on those pins. Again as these bits were toggling our coverage metrics were reporting the pins as covered, only functional qualification exposed the missing checkers.

2) ***Missing Reset Tests and Checks****:* The second most common item uncovered by running functional qualification on all these environments is that we do a poor job of verifying the reset functionality. In almost all cases there were initially no tests that asserted reset multiple times during a test so only a power on reset was being tested. The most common excuse for the missing tests is that they are difficult to write and many testbench monitors could require rework to support random reset assertions.

3) ***Logic Activated but Poorly Propagated*** : The most unsettling item uncovered during our initial forays into functional qualification is the percentage of faults that are classified as non-propagated. These are faults that are activated but the change in behavior caused by these faults never reaches a checker in the testbench. In the worst cases we have observed as high as 30% of faults are classified as NP. Again, these are faults that would typically be reported as covered using code or functional coverage metrics, but functional qualification shows that the code is not truly being verified.

In many cases the number of non-propagated faults can be reduced simply by including more tests in the evaluation; however this is not an efficient method to remove all non-propagated faults. Improving the efficiency of investigating and removing NP faults is an active area of research for us; both formal and improved stimulus generation techniques are being investigated. We currently target having less than 10% of the total faults classified as NP, but this requirement is often waived because of schedule requirements.

4) ***Missing Negative Checks*** : Often the effect of the faults inserted during functional qualification is for features and functions of the DUT to be enabled when they normally would not be. For example, a clock that should only be toggling when a specific signal is asserted will instead be active throughout the entire test when a fault is inserted. We often find that checkers are written to prove that an action happens when it supposed to but there is no check to prove that the action does not happen when it shouldn't.

5) ***Incorrect Checkers*** : This is the category of items we expected to uncover with functional qualification which were simply mistakes in the testbench code itself, and we have found many of them. They include cut and paste errors in control code, badly coded assertions that do not trigger when expected, and simple specification mis-understanding leading to incorrectly coded checks. Just as there are limitless ways for designers to make mistakes in the DUT there are just as many ways for verification engineers to make mistakes in their code, and functional qualification is a systematic unbiased approach that can uncover them.

D. ***Limitations***

The above examples clearly demonstrate why one should employ functional qualification to objectively measure the quality of their verification environment. Before undertaking this effort one should also be aware of the costs to do so; in terms of schedule, compute resources, and engineering resources the cost to complete a full functional qualification analysis can be extremely high.

1) ***Runtime****:* Runtime is the primary limiter of functional qualification and depending on the number of faults placed in a design and the efficiency of the testbench in detecting those faults, the runtime can become extremely large.

Consider the following example where functional qualification is going to analyze a design with 10,000 faults and 200 test patterns that each take 30 seconds using 10 single core compute servers. Equation (1) shows the calculation to determine the runtime for the activate phase, which is this case is 10 minutes.

Activation Runtime =(# tests * test time) / Number of Servers (1)

During the detect stage of the analysis every fault must be considered in isolation and for a fault to be categorized as non-detected all tests that activate that fault must be run. Taking this to the extreme, suppose all tests activate all 10,000 faults and all faults will be categorized as non-detected. The runtime to complete this analysis would be 100,000 minutes or almost 70 days, as shown in equation (2).

Detect Runtime = (# faults * # tests * test time) / # Servers (2)

Consider the opposite extreme which is a perfect verification environment where every fault is detected by the first test run with the fault included. Even in this ideal case the Detect phase runtime will be 500 minutes or 50x longer than a single regression run.

Detect Runtime = (# faults * test time) / # Servers (3)

In addition to testbench efficiency in fault classification the throughput for your functional qualification analysis will also depend heavily on your company's simulator license and compute farm resource availability. We highly recommend the iterative approach described earlier in which the detect phase is limited to identification of a pre-defined number of non-detected faults which are resolved before restarting the analysis.

2) ***Volume of Data****:* The other major limiter to productivity with functional qualification is the sheer amount of data generated by the tool, and just as the runtime puts pressure on compute resources the data generated can tax engineering resources trying to sort through it all.

Consider again the fictional testbench with 10,000 faults and where we now have completed a full pass through the detect phase. At the end of this phase the functional qualification software reports that 9000 of the faults are detected, 990 faults are non-propagated and 10 faults are non-detected. We would classify this as a good result in that 90% of the design is being inspected by the verification environment, but that still leaves 1000 faults that need to be investigated. For each one this process would include understanding the fault, dumping waveforms to observe what effect the fault had on the design behavior, and finally determining why the testbench is not detecting the fault. At the end of this analysis new checks or tests are added to detect the fault if it is determined that the testbench should not be capable of detecting this fault it can be disabled.

We found it takes anywhere from 1 hour to a full day to fully debug and resolve a single non-detected or non-propagated fault. For this example it could take 10 hours to 10 days just too fully analyze the non-detected faults, which still leaves 990 non-propagated faults to analyze. Our current signoff metrics are defined as zero ND faults and for number

of NP faults to be less that 10% of the total faults. However, we have not been able to establish firm signoff criteria as the number of NP faults is often more than we can analyze in a reasonable timeframe. This is still a weakness in our overall verification methodology that we are actively working to improve.

*E. **Lessons Learned***

*1) **No Environment is Perfect***

The very first environment analyzed using functional qualification provided the "light bulb" moment when we realized we must include functional qualification in our verification signoff process going forward. The targeted testbench was a block level constrained-random environment for a DSP Processor Data Address Generator (DAG) block and was chosen because verification was as complete and perfect as we knew how to do.

Code and Functional coverage were both at 100%, a full verification plan had been written and signed off, and regressions had run clean for multiple weeks. When we began the analysis the functional qualification software quickly returned with a ND fault on output pin that asserted only when a specific instruction was sent to the DAG block. We were incredulous at first, as a specific test and dedicated checker had been written specifically for that pin.

We verified that the test was included in the analysis and then discovered that the testbench control code was incorrect. The control statement that enabled the checker had been copied from another section of code and never updated to fire when the correct instruction was issued. In fact, the checker that was specifically written for this pin was never enabled!

Upon fixing the testbench the fault was detectable and while no design bug was uncovered by the corrected checker the realization was there; if a bug had been present in the design we would not have identified the problem.

*2) **When to Start Functional Qualification** :*

Given the valuable information that will be gathered by running functional qualification on one's testbench it can be tempting to start the process sooner rather than later in the project schedule. However, it is best to wait until the testbench is complete, meaning all planned tests and checkers have been written, before attempting to run functional qualification for the first time. Attempting to run before this stage of the project will result in a large number of NA and ND faults related to the checkers you already know you are missing and make it more difficult identify true ND and NA faults in your environment.

We typically end up running functional qualification two times during the life of a project. The first time as soon as the environment is complete and all planned checkers and tests have been written. This analysis identifies any major holes in the verification plan and environment. Then we will run again once RTL is frozen as part of our final verification signoff procedure.

*3) **Set Clear Priorities***

The time to run and the amount of data generated during functional qualification can be extensive, and depending on the compute and engineering resources available it will often not be possible to resolve every potential testbench weakness identified. This can cause difficulty in determining when an evaluation is completed. The best approach we have found is to dedicate a fixed amount of time to running functional qualification, not setting extensive signoff metrics such as 0 NA, 0 ND and 0 NP faults.

We typically allocate two weeks in the schedule for debugging functional qualification issues and our minimum signoff requirement is zero Non-Activated faults and zero Non-Detected faults. If schedule pressure is particularly high we will limit the Non-Detected criteria to connectivity faults only.

*4) **Increasing Qualification Throughput***

Due to the large number of repetitive test runs required for functional qualification, anything that can be done to improve individual test runtimes can have a significant impact on the overall effort. Below are a few items we have identified that can help speed up the process.

Often environments will continue to run tests even after a failure has been identified in an effort to provide more debug information in log files for engineers. During functional qualification this is not required as a single failure is enough to determine that a fault is detectable. Test environments undergoing functional qualification should always be configured to stop after the first failure.

Most tests used in functional qualification activate hundreds or thousands of individual faults in the design and depending on your license availability and compute farm resources many faults can undergo analysis in parallel as long as tests are reentrant. For a test to be reentrant it must be possible to run multiple instances of the test in parallel without those tests affecting each other. An example of a non-reentrant test is one that outputs a file and reads it back during execution. If multiple versions of this test are running the files can be overwritten and create false failures. Environments with non-reentrant tests will run much slower than environments with reentrant tests.

The last recommendation to improve qualification is related to testbench architecture and placement of checkers. Often testbench checkers are placed only on the top level outputs of

the DUT which then requires the effects of faults to be propagated through the entire design hierarchy to be detected. Including assertions in the design code and increasing the usage of block level checkers in an environment can greatly improve efficiency in detecting faults. Including checks throughout the design also promotes faster debug, as failures on top level pins can often be difficult to trace back to their root cause. Anything that can be done to bring checks closer to the faults will greatly improve the efficiency of functional qualification as well as the overall verification quality.

IV. Conclusion

While acknowledging the significant run-times and resources required to complete functional qualification this paper has shown that the effort is required because of the complexity and size of modern verification environments. Without an unbiased, systematic approach to measure the effectiveness of a verification environment there will be holes in your environment which could lead to bugs in your design.

Acknowledgments

I would like to thank Marty Rowe and Myles Glisson of Springsoft/Synopsys for their help integrating Functional Qualification into the Design Verification Methodology at Analog Devices.

References

[1] M. Hampton, "Functional Qualification," EE Times, June 25th 2007.