

Improve functional verification quality with mutation-based code coverage

By George Bakewell

Embedded.com

(12/07/09, 05:39:00 PM EST)

Despite advances in stimulus generation and coverage measurement techniques, existing tools do not tell the engineer "how good" the testbench is at propagating the effects of bugs to observable points or detecting incorrect operation that indicates the presence of bugs.

As a result, decisions about where to focus verification effort, how to methodically improve the environment, whether it is robust enough to catch most potential bugs, and ultimately when verification is "done" are often based on partial data or "gut feel" assessments.

This article discusses the application of mutation-based testing techniques to measure and drive improvement in all aspects of functional verification quality for simulation-based environments as a solution to these problems.

Existing methods

Functional verification consumes a significant portion of the time and resources devoted to the typical design project. As chips continue to grow in size and complexity, designers must increasingly rely on a dedicated verification team to ensure that systems fully meet their specifications.

Verification engineers have at their disposal a set of dedicated tools and methodologies for verification automation and quality improvement. In spite of this, functional logic errors remain a significant cause of project delays and re-spins.

A key reason is that two important aspects of verification environment quality— the ability to propagate an effect of a bug to an observable point and the ability to observe the faulty effect and thus detect the bug— cannot be analyzed or measured. Existing methods such as code coverage and functional coverage largely ignore these two aspects, allowing functional errors to escape the verification process despite excellent coverage scores.

Code coverage at its core is a simple measure of the ability of the stimulus to activate the logic in the design, where "activate" means execute every line, toggle every signal, traverse every path or some similarly discrete activity.

While this is a necessary condition, you can't find a bug if you don't "touch" the code related to the bug—it is certainly not sufficient to expose the presence of all or even most problems in a design.

Code coverage says nothing about the ability of the verification environment to propagate an effect of a bug once activated or detect its presence assuming propagation is achieved. Verification engineers thus accept that while code coverage provides interesting data, it is a poor measure of overall verification environment quality.

Functional coverage is generally more interesting, necessary and useful in its own right. In simple terms, it provides a way to determine if you've exercised all important areas of functionality, where "important" is defined in various ways, such as "all operational states", "all functional sequences" or the like.

The rub is that, by definition, functional coverage is subjective and inherently incomplete. The functional areas (functional coverage "points") to be checked are defined by a set of engineers and typically based on the design specification—a document that thoroughly describes how a design should operate but does not provide a comprehensive view of how it should not operate.

If the specification considered all possible bugs that could exist in the design description and described how they might manifest themselves in terms of function, it would be a simple matter of translating this list into a

set of functional coverage points to be checked comprehensively during verification.

Unfortunately, this is not the case, and while functional coverage provides useful and necessary data, you must have some means of determining if you are verifying the functionality laid out in the specification. Like code coverage, it is a poor measure of overall verification environment quality (**Figure 1 below**).

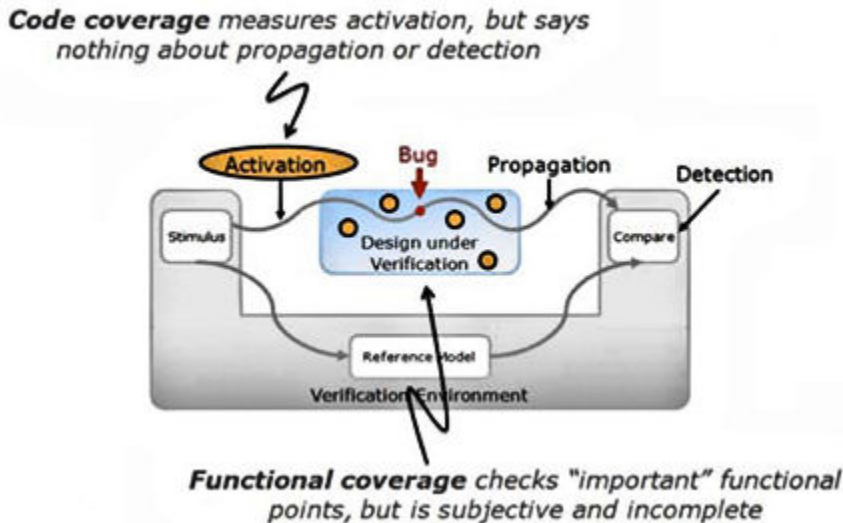


Figure 1: While functional coverage provides useful and necessary data, you must have some means of determining if you are verifying the functionality laid out in the specification.

Mutation-based techniques

The application of mutation-based testing technology may be the key to addressing the shortcomings of existing tools and closing the "quality gap" in functional verification.

Mutation-based testing originated in the early 1970s in software research. In its original application, this technique aimed to guide software testing towards the most effective test sets possible.

A "mutation" is an artificial modification in the tested program, induced by a fault operator. It changes the behavior of the tested program. The test set is then modified in order to detect this behavior change. When the test set detects all the induced mutations (or "kills the mutants" in mutation-based nomenclature), the test set is said to be "mutation-adequate."

Several theoretical constructs and hypotheses have been defined to support the validity of mutation-based testing. In the field of digital logic verification, the basic principle of injecting faults into a design to check the quality of certain parts of the verification environment is known to verification engineers.

Engineers occasionally resort to this technique when they have a doubt about the testbench and there is no other way to obtain feedback. In this case of "handcrafted" mutation-based testing, the checking is limited to a very specific area of the verification environment that concerns the verification engineer. Expanding this manual approach beyond a small piece of code would be impractical.

The automation of mutation-based techniques, however, offers an objective and comprehensive way to evaluate, measure and improve the quality of functional verification environments for complex designs.

Applied intelligently, a mutation-based approach provides detailed information on the activation, propagation and detection capabilities of verification environments, and identifies significant weaknesses and holes that have gone unnoticed by classical coverage techniques.

The analysis of the faults that don't propagate or are not detected by the verification environment points to

deficiencies in stimuli, observability and the checkers that are used to detect unexpected operation and expose the presence of design bugs.

Functional qualification

The application of mutation-based technology to measure the quality of and find weaknesses in the verification of digital logic designs is known as "functional qualification."

This process was pioneered by Certess in 2006 with the Certitude Functional Qualification System. The practical application of the technology involves the three-step process shown in **Figure 2 below**.

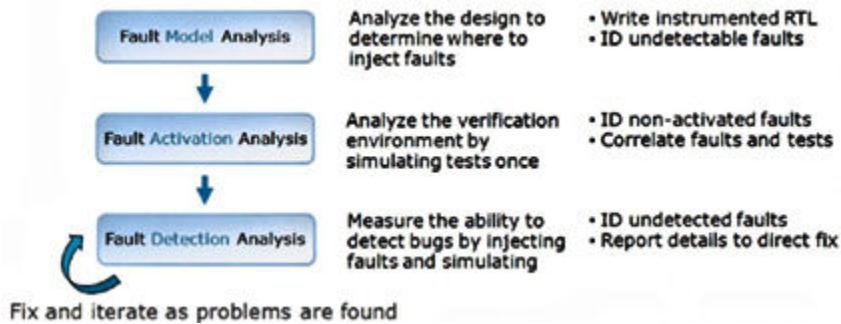


Figure 2: The three-step functional qualification process.

The fundamentals of the process are:

- 1) a static analysis phase that determines where faults can be injected and writes out a new version of the RTL code that enables this injection;
- 2) a simulation step that determines the set of faults that are not activated by any test and—for those that are activated—correlates each fault with the tests that activate it; and
- 3) a detection process that injects faults one at a time, runs the relevant (correlated) tests to determine if a given fault is detected, and in the undetected case provides feedback to the user to help direct the fix (add a missing checker, develop a missing test scenario etc).

The complexity of today's designs and the number of tests required to verify them demands additional automation and intelligence, lest the combination of possible faults and associated tests overwhelms the process and it becomes impractical.

For example, it is likely that certain faults are undetectable, either due to redundant logic or "dead code" that has no path to an output. The identification and elimination of these faults during the static analysis phase can save significant simulation time during the activation and detection steps that follow.

Similarly, information about a given test gathered during the activation phase—simulation time required, number of faults activated, etc.—can be used to great advantage in making the detection process more efficient.

Methodology considerations are also vital to making functional qualification technology practical for today's chip designs. Research has shown that certain classes of faults are more likely to expose significant weaknesses than others, so early identification and qualification of these faults can provide quick feedback on the overall health of the environment.

Fixing problems related to high-priority faults as soon as they are found allows for immediate and significant improvement to the verification environment and provides motivation for applying partial functional qualification early in the verification process.

Incremental usage—running functional qualification, fixing problems and running again— also allows for the accumulation of qualification data over time, since faults that are detected early in the process don't need to be re-qualified later unless the related RTL code changes.

Research has also shown that fixing "big problems" early tends to fix many "smaller" or more subtle problems related to the same missing checkers or test scenarios, so an incremental approach makes the overall process more efficient.

Finally, complete and flexible access to all analysis results is a must. For example, intuitive reports are needed to show where faults have been injected into HDL code, provide fault status, and easily access details about any given fault.

It is most critical that the functional qualification systems be tightly integrated with existing commercial simulators and advanced debug systems and fully compatible with current verification methodologies such as constrained random stimulus generation and assertion-based verification.

Conclusion

The verification of today's leading-edge chip designs is an extremely complex process, and the associated verification environments are often more complex than the designs themselves. The opportunities for problems that let RTL bugs slip through the process are myriad.

Missing or broken checkers that fail to detect incorrect design operation and inadequate test scenarios that exercise the entire design but don't propagate the effects of hidden bugs are the most obvious problems.

Mistakes in the "wrapper scripts" that launch and manage the verification process can be even more troublesome, allowing large sets of tests to be marked as passing when they should fail and potentially masking serious RTL bugs, since engineers are not prone to investigate and debug passing tests.

Current techniques such as code coverage and functional coverage, while providing interesting and sometimes necessary data on verification status and progress, cannot adequately address these issues.

Their incomplete and subjective nature leaves too much room for error and oversight. Mutation-based techniques, on the other hand, provide both a comprehensive and objective assessment of verification environment quality.

The functional qualification process built on these techniques measures the environment's ability to activate logic related to potential bugs, propagate an effect of those bugs to an observable output, and detect the incorrect operation and thus the presence of the bugs.

In doing so, it identifies serious holes in the verification environment that can let real RTL bugs escape the process and provides guidance on how to close those holes. The addition of automation and intelligent operation combined with the right methodology makes the application of functional qualification both practical and accessible throughout the verification process.

George Bakewell is Director of Product Marketing at [SpringSoft Inc.](#)

Please [login or register here](#) to post a comment
or to get an email when other comments are
made on this article