Mitigating verification risk by understanding the coverage

By Nicolae Tusinschi, Product Manager, OneSpin Solutions

The effective testing of an IC during the verification process prior to fabrication remains a perennial issue for design engineers, made worse by the inability to accurately measure the progress of verification.

There are various verification coverage techniques, all with disadvantages, leading to incomplete verification, poor test quality and duplicated verification effort on well-tested parts of the design. Many reasons lie behind these drawbacks, including poorly-defined metrics, incompatible tools and unclear methodologies.

A good verification environment should include all the important functions of a Design Under Verification (DUV) through coverage metrics, to provide a multi-dimensional measurement of the verification progress. For example, the quality of the input stimuli created during verification will determine if certain operational scenarios are triggered (functional coverage), or that certain parts of the DUV code are activated (structural coverage). These are valuable measurements when assessing the quality of the verification process as a whole.

Formal-based processes

Before silicon tape-out, companies usually insist on specified coverage goals being reached in order to qualify the design. There are two potential reasons why a certain operational scenario is not covered: The first is that the verification process doesn't produce the right test patterns, requiring further effort; and the second is that the scenarios remaining uncovered cannot be activated at all, because they are simply unreachable – i.e., can't be stimulated due to the design code structure. Overall, it can be difficult to ascertain which of these situations is occurring.

Formal-based technologies can help resolve these issues, by applying them in a simulation-based flow, generating scenarios due to the exhaustive nature of their operation.

If coverage data is available from other sources, such as simulation output or a coverage database, this automatic test generation process may be guided toward previously-untested scenarios or coverage "holes". At this point, a formal engine can either automatically generate test stimulus for the DUV to cover the hole, or prove that no such stimulus exists, and it is therefore pointless to look for coverage where it is unachievable.

This formal-based process is fully automatic, and in both cases provides valuable information. Ideally, this process would be repeated for each iteration of the design. In this application, the formal technology is being leveraged to assist simulation-based verification. Engineering effort may be reduced significantly without requiring the engineers to acquire formal analysis expertise.

Formal tools are exhaustive by design and consider every potential input scenario. However, often some of these scenarios represent illegal inputs and would generate scenarios that can't happen during device operation, hence should be excluded from the analysis. Formal constraints would then be added to exclude illegal input scenarios. However, if these constraints are too tight and exclude legal input sequences, the tool might not evaluate vital behaviour. The formal constraints must themselves be verified, to ensure the design verification process is not over-constrained.

Running formal coverage reachability analysis in the presence of constraints will provide an indication of code that may not be activated. If the constraints are too tight, some cover goals will become unreachable, highlighting which functionality has accidently been excluded. In this case we call the coverage goal "constrained".

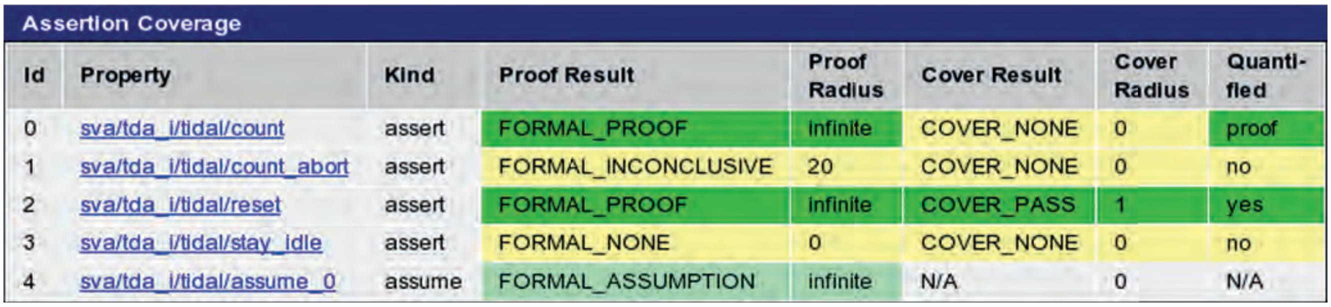
Using formal technologies, we can decide if a scenario is reachable or unreachable. If reachable, the formal tool will provide an indication of how it may be reached, allowing a test to be added to the verification environment. Constrained scenarios need further analysis to avoid accidental over-constraining. In the context of structural source code based metrics, we call this system a "simulation coverage metric", since it relates to the notion of activation in a design common across simulation and formal verification.

Even when a simulation coverage is considered sufficient, there is still a key aspect that must be accounted for, yet is often overlooked.

Simulation-based coverage

In a simulation-based verification environment, checks sometimes implemented as assertions are included to ensure that design behaviour is as expected. In a formal-based environment, all the tests are coded as assertions, which are evaluated against the design code. However, how do we know if we have written enough assertions?

One basic option would be to measure assertion coverage, which is the status of each assertion according to the depth in the design state space it has been proven (proof radius) and whether it was triggered (cover radius); see Figure 1. This way we know that assertions have been written and if they have been



Assertion Coverage

Id	Property	Kind	Proof Result	Proof Radius	Cover Result	Cover Radius	Quanti-fied
0	sva/tda_i/tidal/count	assert	FORMAL_PROOF	infinite	COVER_NONE	0	proof
1	sva/tda_i/tidal/count_abort	assert	FORMAL_INCONCLUSIVE	20	COVER_NONE	0	no
2	sva/tda_i/tidal/reset	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
3	sva/tda_i/tidal/stay_idle	assert	FORMAL_NONE	0	COVER_NONE	0	no
4	sva/tda_i/tidal/assume_0	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A

Figure 1: OneSpin assertion coverage display

run successfully – a useful start. But, there is no relation to the design code behaviour, nor do we know if the assertions are meaningful. It would be more useful to have some kind of metric based on the code itself. Simulation coverage is important, but it is not comprehensive enough to answer these questions, as it is only concerned with the quality of stimuli.

If regions of the DUV are not necessary and can, for example, be modified without affecting the verification, they are either not required or not verified. We need to ask what is necessary to make all the tests and assertions pass. If we measure the amount of observation done with checks and assertions, we can derive a different kind of coverage that is based on observability.

There are several metrics based on the observability principle, such as Cone of Influence (COI) analysis and mutation analysis.

The transitive logic COI is the collection of all signals and expressions potentially having an effect on the value of a signal of interest, subject to a check or assertion. COI analysis finds code locations that are trivially not observed by an assertion because they are not in its COI. However, the COI for some assertions can become very large without providing a good indication of what design sections are actually observed by the assertion. Hence, it is not clear whether any of the code locations in the COI of the assertion are actually observed or not.

Observation coverage

A classic notion of observation coverage is that of mutation analysis. The idea is to define a set of potential modifications to the source code of the DUV that correspond to certain types of errors, a collection of which is called a "fault model". A mutation coverage tool would then apply various mutations defined by the fault model to each potential fault location of the DUV, re-run the verification for each location and each fault, and then test to see if the modification causes a check fail. Obviously, the different fault models lead to different coverage results, rendering a comparison difficult. Various other shortcomings have also been identified:

- Mutations have to correspond to syntactically correct code modifications in order to be applied on the source code. This limits the type of errors that can be modelled.
- Certain mutations at certain locations can render parts of the code unreachable, make assertions vacuous, or both. This is hard to predict and the results become more difficult to understand.
- The number of checks to be run is proportional to the number of checks, multiplied by the number of fault locations, multiplied by the number of error types. Although some process shortcuts are possible, for example restricting the analysis to the COI, they will make the process very expensive in run-time tool requirements.

A good observation coverage metric should have the following attributes:

1. Poorly-written assertions that don't verify anything should not skew coverage results, while additional verification should lead to increased coverage.
2. The measurement method should be independent of syntactic restrictions.
3. The metric should relate back to the source code, so it is easily understood.
4. The measurement method should not lead to unreachable scenarios.
5. The metric computation should be accomplished with a reasonable level of compute resources and run time.

Quantify is a technology available from OneSpin Solutions that makes use of observation coverage to provide an exceptional measurement of code coverage achieved by proven assertions. The solution analyses code statements and branches to provide a concise metric as to whether these statements are observed by a given set of assertions operating under a set of constraints.

Quantify has been proven in real design flows to exhibit a number of advantages over other coverage methodologies as well as other forms of observation coverage.

The problem of accurate verification coverage and its impact on the risk of product failure and schedule overrun is well documented. Coverage solutions to date have proven helpful but inadequate, although some may be improved through formal technologies. Observation coverage takes a different approach, and with the use of advanced formal-based technology, rigorous metrics may be produced that can significantly increase the confidence of effective design verification. EW