

# The Great EDA Cover-up

Brian Bailey

## ***Abstract***

Functional verification is an art, or at least that is what we are led to believe. Every once in a while a new technology emerges that injects a dose of science into the process and these advances can make the process more predictable, increase efficiency and lower overall verification costs. At the same time, the introduction of standards can ensure a degree of commonality in the capabilities provided by the EDA industry. This whitepaper will examine the state of the art in coverage metrics and try to separate some of the hype from reality. It will explore the existing types of coverage metrics, looking at their strengths and weaknesses, and then discuss some technologies that have been developed that can enhance the credibility of the information that coverage metrics provide. While the EDA industry is not yet at the stage of providing a go/no-go indicator informing a team that verification is complete, we are getting to the point where verification can be migrated somewhat from the realm of being subjective to one of being more objective.

## ***Introduction***

Verification methodologies and tools have made significant progress over the past five years as the proportion of total time and money spent on the task has increased. While there is significant disagreement over the total costs and the quality of the results, there is no argument that verification is getting tougher than it used to be. One area of advancement has been the coverage metrics used, and a sign of the importance of this subject is the recent formation of a standards group within Accellera<sup>1</sup> to bring convergence on the metrics and to define an interface that will allow users to combine coverage data of various types and from a number of different sources.

There are two primary roles for coverage metrics: 1) to provide an indication of the degree of completeness of the verification task and 2) to help identify the weaknesses in the verification strategy. The measure of completeness, while often based on objective measures, has traditionally been treated as subjective since most of the metrics in use today can only identify when the task is *not* complete, rather than when it *is* complete. This article will explore the reasons for this and how those metrics can be improved. When the metrics identify an unverified behavior of a design, then changes can be made in the verification environment to enhance the possibility of those behaviors being exercised and verified.

To look for solutions to these issues, it is worth reminding ourselves about the basics of verification that are often forgotten. These will be discussed in the next section, followed by a look at the fundamental types of coverage in use today along with their strengths

and weaknesses. This is followed by a discussion of two recent advances that can provide additional confidence in coverage data.

## ***Verification Fundamentals***

The IEEE definition for verification is:

*“Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled.”*

While not the easiest statement to understand it does contain many of the necessary elements that define a good verification strategy. It stresses the need to be objective and that active examination is required. Unfortunately, the verification process today is still somewhat subjective and verification closure is seen as obtaining the confidence necessary to release a product without significant errors, and with minimum cost. There can be no absolute associated with this goal as different companies and types of product will require differing levels of confidence and have different time-to-market or cost constraints. For example, an inserted medical device requires a higher degree of confidence than a child’s toy. But the amounts of time and effort that can be spent on verification while maintaining the ability to make a profit are just as important, although very different, for the two examples.

Verification is fundamentally the comparison of two or more models, each developed independently, with the assumption that if they express the same behaviors, then there is a high degree of confidence that they represent the desired specification.<sup>ii</sup> The verification model thus provides an objective model (provided that it was developed independently) that, when compared against the design, provides the evidence of fulfillment of the requirements. The IEEE definition does, however, expose a weakness of functional verification because *‘specified requirement’* is subjective and thus it will never be possible, while this is true, to turn verification completely into a science. How do you know that the requirements specified are complete, or that they actually cover the things that are the most important to you?

So the proverbial problem with verification is how to ensure that you have adequately compared the important behaviors represented by both models. The use of intellectual property (IP) was meant to have helped in this regard, but we have not yet attained the position of being able to re-use without re-verification. Until this happens, design and verification do not scale properly. Obtaining a savings in the design phase is of diminishing value if we do not save the same amount in verification. That is why we turn to coverage metrics as a way to quantify the progress and identify the weaknesses in the verification strategy. It is also why the standardization of coverage data is important so that verification data about a piece of IP can be fully integrated into a verification strategy.

There are two primary ways to perform the verification comparison – static or dynamic methods. Static methods, also called formal verification, exhaustively compare properties against the implementation model. Dynamic methods include simulation and emulation techniques which show what the design would do under various forms of stimulation. In the past few years, this methodology has been enhanced by the

introduction of random-based stimulus injection methods. These allow for the pseudo-random sampling of many viable sets of input stimuli, a technique that has been shown to be both efficient and effective in terms of human capital. But is it enough?

The problem with any dynamic verification approach is that it is based on a sampling process and there can be difficulties in getting completeness. Formal verification solves that specific problem, since it performs an exhaustive comparison, but 1) it is not always able to come up with a definitive answer; 2) it is limited in the size of the problem it can tackle; and 3) it is still not possible to ensure completeness because there is no tool to measure formal coverage. Due to these limitations, dynamic verification will continue to be the workhorse of verification, but formal is becoming an essential tool for an increasing portion of the verification process.

In order to satisfy the definition for verification, three things must happen:

- **Step 1 – Activate.** The functionality associated with a specific requirement must be activated. This is a classic controllability issue and many of the existing coverage metrics are closely associated with this phase of verification.
- **Step 2 – Propagate.** The effects of this functionality must be observable. This is a fundamental problem for many verification strategies which check for specific indicators of success. While we may see the correct result in a memory location, we may fail to observe unintended side effects that have not been propagated to a point that is being observed, or that are not propagated at all.
- **Step 3 – Detect.** The propagated results must be compared against the right behavior – as represented by the verification environment. It is fairly rare that full end-to-end checkers exist for the entire design today, but this is becoming more common. In addition, intermediate points can also be verified, but this creates additional work and difficulties for the verification models. Assertions can often be used to provide localized checking.

This three step process of activation, propagation and detection is the basis on which all verification environments are constructed, including formal techniques. It should be quite evident that any efforts spent on behavior activation that are not propagated and detected is wasted effort. But probably even more problematic is that if we believe that a behavior has been verified just because it has been activated, then that provides a potentially false level of confidence. Thus failure to follow the three steps leads to inefficient verification and the potential for overly optimistic results.

The next section looks at common coverage metrics and how they relate to these three steps in the domain of dynamic verification.

## ***Traditional Coverage Metrics***

There are many types of coverage metrics and this is not intended to be an exhaustive review of them. Instead, this article will look at some of the attributes of the different types that exist. For that purpose, coverage metrics have been divided into four categories: 1) Ad hoc, 2) Structural, 3) Functional and 4) Assertion coverage.

## Ad Hoc Metrics

This includes metrics such as bug discovery rates, bug densities and a number of other metrics that provide a general indication about the stability of the design. They provide no objective information about the quality of the design or the verification progress, but they have been a useful management indicator in the past and continue to be used today. While no one would release a chip based on these metrics, a deviation of these from the norm, or from experience, can highlight a project that is getting off track or hitting certain kinds of problem.

## Structural coverage

This is the class of metrics that are related to the structure of the implementation. Most people perhaps know it as code coverage, although this is just one example of structural coverage. Within code coverage, there are many possible metrics such as line, branch and expression coverage. Almost all simulators on the market will offer a number of different types of code coverage. There are a number of problems with these metrics, including:

- They can only cover what has been implemented. Missing functionality is not identified.
- Isolated observations: Just because a line of code was reached, does not mean it was executed for the right reason or that it did the right thing. In other words, it is a metric based on “activation” only.
- Independent of data: Each metric is only associated with control aspects of the design.
- No prioritization: All lines of code are considered equal.

However, in all fairness, these metrics do have a number of advantages, including:

- Cheap and easy to instrument.
- Provides an absolute target. When you have 100%, you know that all of the lines of code, for example, were executed.
- Easy to locate coverage holes, although not always easy to work out how to fill them.
- Provides a good negative indicator. When code coverage is not complete, you know that verification is not complete.

## Functional Coverage

This is a much newer class of coverage metric that is badly named because it does not measure the coverage of functionalities. Instead, it measures the coverage of particular observable events that are an indicator of those functionalities having been executed. The list of coverage points is generally obtained from a specification or verification plan. In many cases, the coverage points are defined incrementally, and this provides a secondary advantage in that the early coverage points are likely to be the most important ones. An example coverage point would be to observe that all data input packets have been seen on each input, or that all possible sequences of data packets have been observed. Functional coverage metrics have a huge advantage over structural metrics in that they can identify missing behaviors. Problems include:

- Can be time consuming to define.
- Incomplete. There is no way to tell if all functionalities are covered.

- Isolated observations: Just because a coverage point was observed does not mean it was seen for the right reason or that the right thing happened as a result. In other words, it is a metric based on “activation” only.

Advantages include:

- Can identify missing functionality since it can represent aspects of the specification that were not included in the design.
- Allows more focus in the verification process – higher priority items are likely to be targeted first. When complete, metrics will be extended to less important areas, etc.

The biggest problem with functional coverage is that it is incomplete. For this reason, many coverage vendors recommend that code coverage is used at the same time in order to help identify the weaknesses in the functional coverage model. Given the problems with code coverage, this is a weak endorsement for the effectiveness of functional coverage.

## Assertion Coverage

This term is somewhat problematic since it means different things to different people. Within the Accellera standards Unified Coverage Interoperability Standard group, four meanings have been identified so far:

1. Coverage using implicit metrics of an assertion.
2. RTL lines that are within the logic cone of an assertion.
3. Functional coverage implemented using an assertion.
4. Density distribution of assertions throughout source code.

If we consider that an assertion defines a set of functionality that corresponds to some -- hopefully equivalent -- functionality in an implementation, then assertions are a form of checker and an integral part of the verification environment. However, this is somewhat disconnected from using assertions as a coverage metric, in that we are then trying to define how much of the implementation is within the domain of the assertion.

Assertions are built out of properties, which are the fundamental cornerstone of formal verification. It would thus be very interesting to be able to answer the question: When have I defined a complete set of properties that cover all specified requirements? That problem has not been solved today, but does make properties and assertions a very interesting area for future developments. Assertions do however have a number of disadvantages including:

- Time-consuming to write and execute.
- Use a different language or set of language features.
- No way to ensure completeness.
- Inability to handle data transformation.

Due to the current limitations of assertions, they are normally used for verifying fairly localized functionality, such as an arbiter or a FIFO. However, assertions and their languages are new to this industry, so we can assume that they will continue to grow in their abilities in the future and overcome many of these limitations. Assertion coverage, on the other hand, is not that useful today.

## Improved Methodologies

The big drawback with both structural and functional metrics is that they are both isolated observations of events that provide no indication that the correct behavior is associated with those observations. In other words, they only perform step one of the three necessary steps discussed above. Because of this, these metrics are often referred to as stimulus coverage metrics. They assume that all errors will propagate to an observable output and that a checker or assertion will correctly determine that something is in error. The coverage metrics in place today do nothing to ensure design correctness. This compounds the fact that coverage metrics may lead to over optimism in the quality of the verification that has been performed.

This article looks at two developments that extend verification beyond the first of the three steps of verification. To understand both developments, it is worth drawing an analogy to the production test world. In this world, the design is the reference model and all chips produced should match that reference. A fault model, called the stuck-at fault model, allows for any wire to be stuck-at 0 or stuck-at 1. It has been shown that there was a reasonable correlation between the actual errors that could occur in the manufacturing process and this fault model. When a fault is injected into the circuit, that fault is said to be detected if it produces a value different from that expected by the reference model. That clearly only happens if a fault effect can be propagated to an output. To prevent bad chip escapes, it is necessary to have very high fault coverage.

## Observability-Based Coverage

The first advancement that was made in coverage was to mimic the production test methodology and incorporate observability into the metric. One such development was OCCAM<sup>iii,iv</sup> which computes the probability that an effect of an error would be propagated to an observable output. It does this in a two stage process. Firstly by simulating a slightly modified version of the design from which it collects a set of results, the reference values, and some additional tag values at the activation points. Then in a second, post-processing step, algorithms are used to determine if the tag would be propagated to an output. In essence, the tags represent an error on the assignment statements. Observability-Based coverage determines if there is a difference between the reference design and the one in which the error was introduced.

This work was commercialized by Synopsys and became Observability-Based Coverage<sup>v</sup> (OBC), a part of the VCS<sup>TM</sup> simulator, but appears to have been discontinued as of their 2006.06 release, just at about the time when they received a patent for the technique<sup>vi</sup>. While it was a considerable advance over the previous metrics, in that it included the propagation step, it still assumed that a suitable checker existed to identify an incorrect result and the correlation between the ability to propagate tags and the ability to propagate errors is not clear. For example, an assignment is considered to have a tag when it is exercised but it is not clear that the current values being simulated would expose an error condition. The existence of a checker was not necessary in the case of manufacturing test because nothing was trying to prove that the results obtained from the reference model were indeed right – only that all chips produced matched the reference. In addition, the error model of bad assignments is somewhat restrictive and there is no known correlation between this and successful chips.

## Mutation Analysis

The second advancement comes from the software world. The injection of an error into software or a design as a method of determining verification effectiveness is known as mutation analysis<sup>vii</sup>. Mutation analysis injects errors into either a hardware or software implementation. These errors are called mutants and represent the simplest functional errors an engineer could make. Mutation analysis is based on two hypotheses: first, that programmers or engineers write code that is close to being correct, and second, that a test that distinguishes the good version from all its mutants is also sensitive to more complex errors.

By modifying the design and seeing if a difference is detected, mutation analysis provides a more rigorous approach to fault effect propagation than OBC. Any fault that goes undetected is an indication that either the stimulus set is not complete or that the effects of the stimulus failed to be propagated to an output. It thus manages to complete two of the three steps necessary for verification to be achieved.

A lot of work has been done to identify the kinds of errors that an engineer typically makes and from that derive some error models<sup>viii</sup>. The error models are the finest level of granularity of a change in the design, such that while a design fix may change many things on many different lines of code, we can expect that if a verification environment can detect the individual changes, then it is also highly likely to detect a complex mix of them. Early attempts to prove the value of mutation analysis concentrated on the actual errors made by an engineer rather than isolating the individual components of errors. Because of this, the total number of errors that had to be investigated became very large and took a long time for the analysis to be performed.

Almost all designs have two aspects to them: a datapath and some control logic, so if faults are injected into each of them, then they should be representative of many of the other faults as well. This can help to alleviate the explosion in potential errors. Other analysis<sup>ix</sup> has shown that it should not be necessary to increase the number of errors more than linearly with design size.

Mutation analysis draws on some of the good features of the older coverage metrics, such as structural coverage. The injection of an error model is performed automatically, so there is no user intervention required -- unlike functional coverage and assertion coverage -- and mutation coverage can be automated. Automation makes it quick, effective and objective. Mutation analysis also provides a definitive and objective measure of completeness since a complete analysis of the source will identify the totality of the faults that should be injected and we know what 100% means. Finally, because the mutants may introduce new functionality that was not present in the source, it may identify functionality that is missing, which removes one of the big limitations of structural coverage.

So how much compute time and power does the use of this technique consume? This process is very similar to fault simulation, where it is impossible to say how much longer it will take compared to a fault-free simulation since some faults will be detected very quickly, while others may take longer.

## ***A new approach***

Recently a startup called Certess has announced a product, called Certitude<sup>x</sup>, which further extends on these concepts by adding the third part of the verification process – namely detection. The tool not only looks for a difference in the observed design outputs, as is done by OBC and mutation analysis, but also looks to see if a checker is actually present to detect the difference. They call this new technique *Functional Qualification*, a name reflective of the fact that it provides a direct measure of the quality of the verification environment to detect functional errors.

The tool has the ability to identify missing checkers, which can both improve the verification environment and also help to reduce the total time spent in verification, since each test will become more effective. The ability of a test to find errors is a better indicator than its ability to increase some activation-based coverage metric. If a set of tests can find all of the injected errors, then, by definition, they will provide good coverage of the functionality. In addition, if we assume that the checkers are developed independently, then the injection of errors may create new functionalities. Since these new functionalities could go undetected and thus flagged, this technique can also find missing functionality.

Certess received a surprising number of comments in the recent DeepChip verification census<sup>xi</sup>. While most of the comments were anonymous, they almost all spoke about the tool finding bugs and inefficiencies in their testbenches, and particularly in checkers that were missing. If the tool even comes close to living up to its early promise, it will enable verification to become more objective and push verification towards science rather than art.

A discussion with Mark Hampton, the CTO of Certess also revealed some interesting insights. They first perform the fault injection process to identify the total number of faults. Then they do a fault-free simulation to find which tests activate which faults. Some faults will be activated by few tests, an indication that these are more difficult aspects of the functionality to verify. In early analysis runs, they will then target these faults first – in other words they will intelligently select the faults that are the most likely to identify verification weaknesses.

In a recent panel that I moderated, Michael Benjamin, the functional verification group manager of ST, talked about the problems they are having today with the lack of adequate metrics to measure quality. He said that Certitude provided “an objective and quantifiable coverage metric for functional verification.” This has helped ST make informed decisions about the verification process and the quality of their IP verification. In a Certess press release, Jean-Marc Chateau, programmable products group director of R & D for ST said, “We are now able to measure with confidence the efficiency of the functional verification of the IP blocks integrated in our SoCs.”

This is clearly another big step forward in the path for objective verification and is the first to address all three stages of a complete verification environment.

## Conclusions

Almost all of the coverage metrics in use today are fundamentally flawed in that they only check the activation aspect of a verification environment. While significant improvements have been made in these metrics, they have failed to address the fundamental issue, which is that both observability and detection must be present before any actual verification is being performed. We can continue to pull the wool over our eyes, or start looking for solutions such as those attempted by Synopsys, or the advances in mutation analysis which extended coverage to include observability, or, more recently, the introduction of functional qualification by Certess that includes the third aspect necessary for complete verification, that of detection. Only with a complete objective view about the verification process will the industry be able to reduce the number of chip failures being reported today and minimize the amount of time and effort that needs to be spent in verification. Now if only we could make the '*specified requirements*' from the IEEE definition of verification objective, then we would have turned what today is an art into a science.

## References

- 
- <sup>i</sup> Accellera UCIS website <http://www.accellera.org/activities/ucis>
  - <sup>ii</sup> Brian Bailey, *How many models does it take*,  
<http://ElectronicSystemLevel.com/phpBB/viewtopic.php?t=8> Accessed 05/2007
  - <sup>iii</sup> Devadas, Ghosh and Keutzer, *An Observability-Based Code Coverage Metric for Functional Simulation*, ICCAD 1996
  - <sup>iv</sup> Fallah, Devadas and Keutzer, *OCCOM—Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification*, IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, VOL. 20, NO. 8, August 2001
  - <sup>v</sup> Reynaud, *Code Coverage techniques – a hands-on view*, EE Times 09/12/2002
  - <sup>vi</sup> US Patent 6990438 issued January 24<sup>th</sup> 2006
  - <sup>vii</sup> DeMillo, Lipton and Sayward. *Hints on test data selection: Help for the practicing programmer*. IEEE Computer. (Apr) 1978.
  - <sup>viii</sup> Zhang and Harris, *Mutation Analysis for the Evaluation of Functional Fault Models*, HLDVT 1999
  - <sup>ix</sup> Campenhout, Al-Assad, Hayes, Mudge and Brown, *High-Level Design Verification of Microprocessors via Error Modeling*, ACM Transactions on Design Automation of Electronic Systems, Vol. 3, No. 4, October 1998
  - <sup>x</sup> Certess press release: *Certess Launches Certitude, First Product to Enable the Functional Qualification of Electronic Designs*, 05/07/2007
  - <sup>xi</sup> Deepchip website: <http://www.deepchip.com/items/dvcon07-05.html> accessed 05/2007