Today, most design managers depend on some sort of verification-coverage metrics to answer three primary questions, according to Mentor Graphics’ chief verification scientist, Harry Foster: Where have I been, where am I going, and when will I get there?

But there are many coverage metrics, coming from various tools and meaning different things. It is vital for design managers to understand what each sort of metric really means. Equally important, the manager must be able to blend a variety of metrics into one picture that will answer Foster’s three questions. Most important, the manager must answer correctly another version of the third question: When is it time to stop? The decision not only requires a fusion of diverse metrics, but also depends upon a detailed verification plan that has existed since the early days of the architectural design and has since grown and blossomed right along with the design.

The idea of code coverage, which designers borrowed from identical tools in the software-verification world, is simple: As you run RTL (register-transfer-level) simulation, you simply maintain a 1-bit-wide table with an entry for each line of code in the RTL source. At the beginning of a simulation run, you set the bits in the table to zero. The first time you execute each line, you set the corresponding bit in the table to one. When you end the simulation, you have a map of which lines of code you executed and which you did not. If you did not execute a line, it is safe to say you did not verify it.

Experts generalize the notion of code coverage to lots of other implicit measures of coverage based on the RTL view of the design. You can devise tools that report on coverage of RTL expressions, branches, or toggling of registers. And most such reports are readily available from commercial simulation tools without your having to devise special monitors.
The early appearance of code-coverage metrics and their ease of use have made them popular. Foster says that Mentor’s survey data indicate that about half of design teams have code-coverage metrics somewhere in their verification flow. But there are serious issues with code coverage, as well. The primary one, says Synopsys fellow Janick Bergeron, is that “structural coverage metrics are necessary, but they are not sufficient to determine verification coverage.” Bergeron points out that the most glaring issue with code coverage is a logical one. The fact that you have executed a line of RTL doesn’t imply that it did what you intended.

More precisely, the problems are of observability and completeness. When you executed this line of code, did its results travel to a node that you were actually observing during this simulation run? If not, then you have no idea whether the code did what you intended. “We have seen designs that have 100% line coverage, but, in fact, the actions of only 70% of the lines were observed during simulation,” Foster says.

Completeness is a separate issue. You executed the line of code. But did you execute it in all of the cases in which it can be active? How about the one case in which it doesn’t work?

**Functional metrics**

These shortcomings have led many teams to use functional verification. Functional coverage asks how many of the functions of the design you have shown to do what you intended. In its intuitive forms, it is the earliest means managers have used to measure verification, and it is the mainstay of many teams.

Hans Sahm, technical manager in hardware research and development at the optical division of Alcatel-Lucent, describes a modern version of this seminal approach. “We start with a requirements document, and we use internally developed scripting to generate a verification-plan spreadsheet from the requirements,” Sahm says. “That spreadsheet lists functional requirements, described in English, and the test cases the verification team has chosen to verify each requirement.” This spreadsheet gives the verification team a single document in which they can check off test cases as they run in simulation and thereby have a function-by-function chart of verification progress.

This concept forms the backbone of all forms of functional-coverage metrics, but it suffers from some serious challenges. As Sahm points out, “There is no automatic link between a functional requirement and the test cases necessary to verify it.” Understanding a requirement and translating it into tests that adequately cover that requirement depends on the skill and experience of the verification engineer.

“There is no automation for thinking,” says Mentor’s Foster.

“There is always a problem with interpreting the requirements documents,” says Jeff Fox, principal verification architect at Altera. “Different engineers can read the same document and come away with quite different ideas of how the function should behave. That is why we try to keep our requirements documents as close as possible to executable code. They must be precise.”

Synopsys’ Bergeron agrees. “When you create directed tests to verify a function, it is an open-loop process,” he observes. “You can never be sure from the result that there isn’t a bug in the test.”

The most common technique for combating this reliance on human frailty has been the use of assertions and constrained random testing, as Verisity, which Cadence now owns, originally championed. According to Mentor’s survey data, only about 40% of verification teams are using constrained random tests. Correspondingly, about 40% are using functional-coverage metrics. Since
the early days, a number of specialized languages have appeared in which to write assertions, but the industry now appears to be converging on System Verilog for this purpose. So, we are seeing a new pattern: assertions in System Verilog, constrained random tests to test the assertions, and verification metrics expressed as coverage of the assertions.

This process is evolutionary for many design teams. Giri Raju, general manager of the semiconductor- and solutions-business unit at Wipro Technologies, describes the path his design teams are taking. “Previously, we had used only code-coverage metrics, tracked in a manual cross-reference table, to manage verification,” he says. “Our goal was simply 100% code coverage. In stages, we have moved to functional-verification tools, and we have continued to track the progress of verification with the manual tables. Now, we are moving to System Verilog and the Open Verification Methodology.

“There is still much engineering skill required. Verification engineers identify coverage points for verification, and review them with the design engineers as a check. We believe we can automate 80 to 90% of this process, but there will always be certain scenarios where things have to be worked out manually. Assertions really help us, though. Our design engineers are now used to inserting assertions in their code.”

One great boon to the process of generating assertion-coverage metrics has been the use of FPGAs (field-programmable gate arrays) for logic verification in a System Verilog environment. Newer tools allow verification engineers to generate constrained random stimulus patterns, and the tools then track hits on coverage points. FPGAs can enormously accelerate this process, Altera’s Fox says, by allowing the team to synthesize the design and the assertions and run tests at or near actual real time. This approach allows the constrained random test creator to cast a much wider net, exploring for not just known but also unknown corner cases.

It also allows for a physical sort of transaction-level coverage. Using an FPGA vehicle, verification engineers can check the operation of an interface, for instance, by simply connecting the FPGA to a known-good external device and watching the transactions. One line of reasoning goes that, if the interface “plays well” with other chips, it is 100% covered.

**Formal tools**

With code-, functional-, and assertion-coverage metrics, the verification manager has many numbers that suggest the degree of completion of verification. Yet, all of them leave some questions unanswered. Increasingly, some teams are using formal-verification tools as adjuncts to simulation-based techniques. These tools bring their own unique metrics to the table.

“On the most critical modules, we are using formal property-checking tools now,” says Alcatel-Lucent’s Sahm. “This [approach] introduces the notion of property coverage. Actually, the tool we use has its own internal completeness-checking capabilities that measure such things as coverage of the properties, and whether the state of each output is determined in a given scenario.”

In a way, formal tools give the ultimate insurance. At the end of a run, you have before you all of the counter-examples that violate each property you specified. By definition, the property is 100% covered. But there’s the question of how completely the set of properties covers the design intent. And, once again, you are back to the skill of the designers, now perhaps reduced because of the notoriously difficult learning curve for the formal-verification tools.
Fusing the data

As you can see, there is no solid answer to verification coverage. Individual tools can tell you how completely they traversed the structure of the RTL code or how fully they checked the assertions the design and verification teams wrote or proved the properties the formal-verification expert defined. But a human process of interpretation and creation separates each of these metrics from the design intent. So, many design managers form their picture of verification progress by fusing the coverage metrics from many sources.

“Different teams have different ways of combining the coverage data into a single picture,” Foster says. “The CPU guys frequently will use only functional-coverage metrics, but they have the resources to make that [approach] work. Without the resources, some design teams still use only line coverage. But you can combine the different kinds of numbers, as well.”

Foster suggests that a team could start with functional-coverage metrics. As the functional coverage approaches 100%, the team could turn on code-line coverage as a check of completeness. This approach, as Altera’s Fox points out, allows the team to spot holes in the functional coverage. If a block of code does not get executed, either it is dead code—which the design team should be able to determine by inspection—or there are some functions of the design that are not covered. “At that point, you write some directed tests,” Fox says.

Fox emphasizes the importance of having data from different sources. “For instance, we were working on an interface IP [intellectual-property] block recently,” he says. “We brought in third-party verification IP from three vendors and put two internal teams on the verification process, as well. Combining the data from all of them found that each approach had overlooked some things.”

Finding the end

Given experiences such as Fox’s, when can a manager say that verification is complete? The cynic would say verification is never complete. The stoic would reply that verification is complete when you overshoot the schedule. The pragmatist has a more interesting answer, however. “You are never ‘done,’” Bergeron says. “But you can reach a level of confidence in the functions that are required for commercial success. It’s a risk-management question.”

Accordingly, Bergeron and Foster say, experienced verification managers watch the coverage metrics from various sources. Commercial tools are available to assist this process by organizing the various metrics by structural block or by function, so verification engineers can look at all the metrics for one area of the design. And efforts exist to ease the fusion process (see sidebar “UCIS ensures interoperability”). Discrepancies at this level usually indicate holes in the verification plan that the team should plug with manually created directed tests.

But the manager should also look at the frequency of bug reports. If bug-report frequency is dropping as the coverage metrics near 100%, all is well. If the bugs keep showing up at a constant rate—or worse—then something is wrong. But when do you stop? Most managers agree: You stop when you are virtually certain of the critical blocks. Sahm defines “critical” as blocks containing entirely new functions, blocks that you cannot easily work around in software, and blocks with which the designers lack experience.
“It is a very legitimate strategy to use coverage metrics to attempt to confine the risk in the design to fixable blocks,” Foster says. “These blocks may have obvious software workarounds. They may have a bunch of uncommitted gates—we used to call them ‘happy gates’—that the designers can use to make patches with just a couple of metal layers. Or the blocks may implement new algorithms that the designers can simply switch off if they don’t work.

“Coverage metrics can’t tell you on what date you will be done,” Foster concludes. “You can watch the coverage curves. If they are flattening out short of 100%, you can change your strategy. If there are big holes in coverage, you can aim your efforts at them. If there are lots of little holes, you may need to change your strategy altogether and maybe try formal methods or a special testbench for the scattered areas. But you watch the metrics to see where you are going, and where you should go next.”

**UCIS ensures interoperability**

Today's contemporary verification flows often employ a diverse set of verification tools, ranging from various forms of simulation to formal verification and even emulation. Each of the processes might generate multiple coverage metrics, which you use to measure the effectiveness of a process and highlight shortcomings that might require attention.

The issue with today's flow is that any tool within a process might generate coverage metrics that are disjointed, overlapping, or a subset of metrics that a different tool might generate. Further, each tool generates its coverage data in a vendor-specific format. Managing this diverse set of coverage data for analysis can be a nightmare for even the most sophisticated project team.

Accellera created the UCIS (Unified Coverage Interoperability Standard) to ensure interoperability when gathering, merging, and interpreting coverage results across a multitool heterogeneous verification flow (Figure A). By using the future Accellera UCIS API (application-programming-interface) standard, multiple verification tools will be able to access a UCDB (unified coverage database), which you can view conceptually as single repository of all coverage data.

**Author's Biography**

*Harry Foster is chief verification scientist at Mentor Graphics.*