Coverage Metrics for Functional Validation of Hardware Designs

Serdar Tasiran
Compaq Systems Research Center

Kurt Keutzer
University of California, Berkeley

Software simulation remains the primary means of functional validation for hardware designs. Coverage metrics ensure optimal use of simulation resources, measure the completeness of validation, and direct simulations toward unexplored areas of the design. This article surveys the literature, and discusses the experiences of verification practitioners, regarding coverage metrics.

CORRECTING FUNCTIONAL ERRORS in hardware designs can be very costly, thus placing stringent requirements on functional validation. Moreover, validation is so complex that, even though it consumes the most computational resources and time, it is still the weakest link in the design process. Ensuring functional correctness is the most difficult part of designing a hardware system.

Progress in formal verification techniques has partially alleviated this problem. However, automated methods invariably involve exhaustive analysis of a large state space and are therefore constrained to small portions of a design. Methods that scale to systems of practical size require either formal, hierarchical design descriptions with clean, well-defined interfaces or considerable human effort after the design is completed. Either way, applying these methods to complex circuits with multiple designers is difficult. Software simulation's computational requirements, on the other hand, scale well with system size. For this reason, and perhaps because of its intuitive appeal, simulation remains the most popular functional validation method.

Nevertheless, validation based on simulation can be only partially complete. To address this incompleteness, simulation-based semiformal methods have been developed. These methods exert better control over simulation by using various mechanisms to produce input stimuli and evaluate simulation results. Validation coverage, however vague the concept, is essential for evaluating and guiding such combinations of simulation-based and formal techniques. The ideal is to achieve comprehensive validation without redundant effort. Coverage metrics help approximate this ideal by

- acting as heuristic measures that quantify verification completeness, and
- identifying inadequately exercised design aspects and guiding input stimulus generation.

Coverage analysis can be instrumental in allocating computational resources and coordinating different validation techniques.[1,2]

Although coverage-based techniques are routinely applied successfully to large industrial designs,[3-5] increasing design complexities have led to renewed interest in this area. In this article, we summarize existing work on cover-

0740-7475/01/$10.00 © 2001 IEEE

age metrics, report on industrial experiences in using them, and identify the strengths and weaknesses of each metric class and of coverage-based validation in general.

Coverage analysis: imprecise but indispensable

One principal use of coverage analysis is to measure the validation effort's adequacy and progress. Ideally, increasing the coverage should increase confidence in the design's correctness. Direct correspondence between coverage metrics and error classes should ensure that complete coverage with respect to a metric will detect all possible errors of a certain type. The lack of well-established formal characterization for design errors makes ascertaining such correspondence difficult. Unlike manufacturing testing, where practical experience has shown that stuck-at faults are a good proxy for actual manufacturing defects, no canonical error model achieves the same for common design errors.[3] Design errors are less localized and more difficult to characterize, making it difficult to establish a formal error model—a common denominator—for even subsets of design bugs. As a result, there is, at most, an intuitive or empirical connection between coverage metrics and bugs, and the particular choice of metrics is as often motivated by ease of definition and measurement as by correspondence with actual errors.

Designers typically use a set of metrics to measure simulation-based validation progress, starting with simple metrics that require little effort and gradually using more sophisticated and expensive ones. Giving a formal meaning to "more sophisticated" also proves difficult. Even if metric M_1 subsumes metric M_2, the input stimuli that achieve complete M_1 coverage are not necessarily any better at detecting bugs than input stimuli that achieve complete M_2 coverage. (Metric M_1 subsumes metric M_2 if and only if, on any design when a set of input stimuli S achieves 100% M_1 coverage, S also achieves 100% M_2 coverage.) To make matters worse, a practically useful, formal way of comparing coverage metrics has not yet been devised.[6] Except for trivial cases, no metric is provably superior to another. In the absence of useful formal relationships between coverage metrics, a comparative measure of "good at uncovering bugs" also needs to be established intuitively or empirically.

Despite these drawbacks, coverage metrics are indispensable tools for simulation-based validation. Input stimuli guided by coverage information commonly detect more bugs than conventional simulation and handwritten, directed tests.[3-5] Moreover, coverage metrics provide more measures of verification adequacy than bug detection statistics alone.[7] At the current state of the art, no set of metrics has emerged as a de facto standard of how much simulation is enough. However, as design groups accumulate experience with certain metrics applied to particular design classes, criteria are evolving to assess how much simulation is not enough.[8] It is in this capacity, in addition to guiding simulation input generation, that coverage metrics have found their most widespread use. Because metrics play a crucial role in functional validation, extensive studies are needed to correlate classes of bugs and coverage metrics. Despite some preliminary work,[3,9] progress in this area is far from satisfactory. Few design groups today feel confident that they have a comprehensive set of metrics.

Besides corresponding well with design errors, another important requirement for coverage metrics is ease of use:

- The overhead of measuring a coverage metric should be tolerable.
- Generating input stimuli that improve coverage should be possible with reasonable effort.
- Only a minimal modification to existing validation tools and flows should be required. Most design teams have invested considerable time and money in existing simulators, simulation analysis tools, and test suites.

Unfortunately, direct correlation with design errors and ease of use are somewhat conflicting requirements. The former often calls for more designer input for defining coverage goals, more effort for stimulus generation to achieve coverage, and more overhead for coverage measurement. Each coverage metric is a compromise.

Classification of coverage metrics

Hardware designs are described by myriad representation formats at various abstraction levels. Each format accentuates a particular design aspect and offers unique insight into structure and functionality. Therefore, we classify validation coverage metrics on the basis of the description format for which they are defined.

Code coverage

Code coverage metrics, largely derived from metrics used in software testing, identify which structure classes in the hardware description language (HDL) code to exercise during simulation. The simplest such structure is a line or a block of code (a sequence of lines with no control branches). More sophisticated code-based metrics used in hardware verification are branch, expression, and path coverage. These metrics involve the control flow through the HDL code during simulation and are best described by the control flow graph (CFG) corresponding to that code. Control statements (the statements from which control can jump to one of several places) constitute the branching points in the CFG—for example, **`if`**, **`case`**, or **`while`** statements. Consider the following Verilog fragment:

```
1: always @(oneHot or a or b or c)
2:   begin case (oneHot)
3:     3'b001: z <= a;
4:     3'b010: z <= b;
5:     3'b100: z <= c;
6:     default: z <= 1'bx;
7:   endcase
8:   if (a | b)
9:     d = d1 + d2;
10:  else
11:    d = d1 - d2;
12:  end
13: end
```

The **`if`** statement on line 8 has **`(a | b)`** as the control expression. Branch coverage requires exercising each possible direction from a control statement. For the **`if`** statement, lines 9 and 11 must both be executed during a simulation run. Similarly, for the case statement, lines 3, 4, and 5 must be executed. A more sophisticated metric, expression coverage, requires exercising all the possible ways that an expression can yield each value. For instance, for the control expression **`(a | b)`**, in addition to the case where **`a = 0`** and **`b = 0`**, we must exercise the two separate cases where the expression gives **`1`** because **`a = 1`** and **`b = 1`**.

Path coverage refers to paths in the CFG. For instance, in the Verilog example, the branch of the **`case`** statement on line 4 followed by the **`else`** branch of the **`if`** statement defines one path through the CFG. Exercising all paths may be impossible (because of **`while`** statements, for instance), so a representative subset may be chosen.[10] For example, verification engineers can use the linearly independent paths (a set of paths through a CFG such that each path contains at least one edge not included in any other path) borrowed from software testing to select a subset of paths approximately the size of the CFG. Selecting the subset of paths is a heuristic, design-dependent process.

Measuring code coverage requires little overhead, and because of the ease of interpreting coverage results, these metrics are useful for writing test cases to improve coverage. Almost all design groups use some form of code coverage, and many commercial, proprietary tools exist for measuring and interpreting it. However, unlike the case with software, achieving complete code coverage for hardware is a minimum requirement. More important, hardware designs are highly concurrent. More than one code fragment is active at a time, thus fundamentally distinguishing HDL code from sequential software. Code coverage metrics do not address this essential hardware characteristic. Consequently, requiring complete code coverage for hardware, although necessary, is far from sufficient.

Metrics based on circuit structure

The simplest circuit-structure-based metric is toggle coverage: Each binary node in the circuit must switch from 0 to 1 or 1 to 0 at some point during the simulation. This metric identifies physical portions of the circuit that are not properly exercised.[5] Separating circuits into data path and control, as shown in Figure 1, is useful for defining more sophisticated metrics in this

class. In the data path portion, registers deserve special attention during validation. Each register must be initialized, loaded, and read from, and each feasible register-to-register path must be exercised. (Exercising some physical paths connecting registers may not be logically possible.) Counters often cause errors in the data path, so designers should check whether counters are reset and whether they reach their minimum and maximum values. The control and status signals at the data-path and control interface characterize the communication that takes place between these two circuit parts. Exercising all combinations of assignments to these signals can help uncover bugs.[11]

Metrics based on the circuit structure, like code coverage metrics, are easy to measure and intuitive to interpret. For test generation, knowing the structures that are not exercised is usually sufficient. However, unlike simpler code coverage metrics, exercising certain structures or combinations of signals might not be possible, and deriving this information accurately and automatically is difficult. The common solution is to ask for user input, but this approach may be time consuming and error prone. Eliminating false negatives (coverage warnings about structures that cannot be exercised) is the greatest challenge facing automated tools. As with code coverage, circuit-structure-based metrics provide a lower bound on the amount of required simulation.

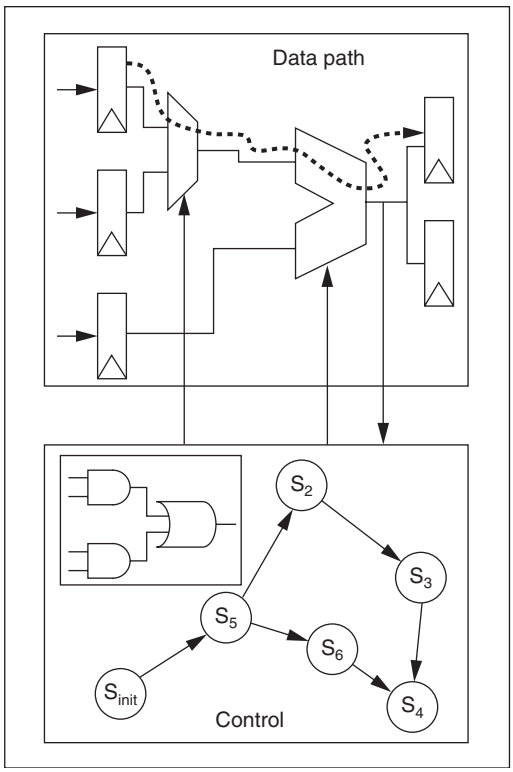

Figure 1. Separation of data path and control for metrics based on net lists.

Metrics defined on finite state machines

Metrics based on code and net lists are defined on static, structural representations; hence their ability to quantify and pose requirements on sequential behavior is limited. Metrics defined on state transition graphs are more powerful in this regard. These metrics require state, transition, or limited path coverage on a finite state machine (FSM) system representation; see the control FSM of the Viper processor in Figure 2 (page 41), for an example. Some control portions are better represented by a collection of interacting FSMs. In this case, using metrics defined for multiple FSMs makes sense. For instance, the pair-arcs metric requires exercising all feasible pairs of transitions for each pair of controller FSMs. In a system with two Viper processors, one pair arc would correspond to one processor making the transition from the exec state to the wr_mem state, while the other makes the transition from start to fetch1.

Because FSM descriptions for complete systems are prohibitively large, these metrics must be defined on smaller, more abstract FSMs. We classify FSMs into two broad categories:

- Hand-written FSMs that capture the behavior of the design at a high level.
- FSMs automatically extracted from the design description. Typically, after a set of state variables is selected, the design is projected onto this set to obtain an abstract FSM (see Figure 2).

Metrics in the first category are less dependent on implementation details and encapsulate the design intent more succinctly. However, constructing the abstract FSM and maintaining it as the design evolves takes considerable effort. Moreover, there is no guarantee that the imple-

mentation will conform to the high-level model. Despite these drawbacks, specifying the system from an alternative viewpoint is an effective method for exposing design flaws. Experience shows that using test scenarios targeted at increasing this kind of coverage has detected many difficult-to-find bugs.[5,12]

The state variables of the abstract FSMs for metrics in the second category can be selected manually or with heuristics. Shen and Abraham present a heuristic technique for extracting the control state variable that changes most frequently, called the primary control state.[13,14] They compute an FSM reflecting the transitions of the primary control state variable and require coverage of all paths of a certain length in this FSM. (Arcs in this FSM are marked by conditions on the inputs and state variables in the fan-in of the primary control state.) Even small processors have a large number of such paths, but because each simulation run is short, the cost is tolerable. Kantrowitz and Noack use transition coverage on a hand-constructed abstract model of the system, as well as cache interface logic.[5] Others select important, closely coupled control state variables based on the design's architecture.[12,15,16]

Selecting abstract FSMs requires compromising between the amount of information that goes into the FSMs and the ease of using the coverage information. The relative benefits of the choice of FSMs and the metrics defined on them are design dependent. Increasing the amount of detail in the FSMs increases the coverage metric's accuracy but makes interpreting the coverage data more difficult. If the abstract FSM is large, attaining high coverage with respect to the more sophisticated metrics like path coverage is difficult. Nevertheless, designers may need to consider paths rather than only states or transitions to ensure that important sequences of behavior are exercised. On the other hand, for designs with a lot of concurrent control where rare FSM interactions can cause difficult-to-find bugs, simple metrics that refer to groups of controller FSMs may be more appropriate.

The biggest challenge with state-space-based metrics is writing coverage-directed tests. Determining whether certain states, transitions, or paths can be covered may be difficult. The FSMs' state variables may be deep in the design, and achieving coverage may require satisfying several sequential constraints. Moreover, inspecting and evaluating the coverage data may be difficult, especially if the FSMs are automatically extracted. Some automated approaches involve sequential testing techniques.[17] Others establish a correspondence between coverage data and input stimuli using pattern matching on previous simulation runs.[13] The capacity of automated methods is often insufficient for handling coverage-directed pattern generation on practical designs, whereas the user may need to understand the entire design to generate the necessary inputs. Nevertheless, state-space-based metrics are invaluable for identifying rare, error-prone execution fragments and FSM interactions that may be overlooked during simulation, thus justifying the high cost of test generation. Ultimately, carefully choosing abstract FSMs can alleviate many of the problems mentioned.

Functional coverage

This category consists of metrics that refer directly to the computation performed by a system rather than its structure. Functionality-based metrics are typically specific to a family of designs and have the form of carefully constructed lists of error-prone execution scenarios and functionality fragments.[3,9] During simulation, each scenario and functionality fragment must be exercised. Larger execution fragments, such as certain instruction sequences for processors or transaction sequences on a bus, may need to be exercised several times.[9] Coverage tools then report the number of times each such case is exercised.

Error-prone scenarios are either specified manually (for example, all pairs of concurrent transactions) or synthesized by tools that search for some predetermined structures in a net list or RTL code (for example, to exercise pipeline hazards, back-to-back instructions that use the same resources). For each case identified, a monitor (a piece of code that runs along with the simulator) is constructed. Designers use a custom language or a test-bench authoring tool to specify this monitor. During simulation, the monitor's state indicates whether the error-

prone case was exercised and whether an error was detected. Monitors can look for simple errors, such as transitions to an illegal state in an FSM, or more complicated event sequences, such as violations of a bus protocol.

An assertion that annotates the design description can be considered a special case of a monitor. Regardless of the validation methodology, user-defined monitors are extremely helpful because they capture design intent expressed at the implementation level and assumptions about the design's environment. Both of these are missing from high-level specifications and are difficult to address using generic structural coverage metrics.[4,5,18]

Following the terminology Grinwald et al.[9] use, we classify metrics that refer to functionality fragments as either snapshot or temporal tasks. Snapshot tasks are conditions on the variable values in a single clock cycle, such as the number and locations of instructions in a processor's branch unit. Temporal tasks involve a series of conditions spanning several (not necessarily consecutive) clock cycles—for example, a 3-to-6-clock-cycle sequence containing an instruction-interrupt-instruction pattern in a microprocessor. Coverage tasks, like error-prone scenarios, are defined with a custom language or a test-bench authoring tool that has temporal operators (mechanisms to specify behavior in future or past clock cycles) and support for the operations (such as arithmetic or logic operations) that the design performs.[9] Code that checks coverage tasks can track the number of exercised times states, state-event or state-command combinations, or event sequences, and this information can help direct later simulations.[5] To decrease the number of cases to test, designers commonly group functionally similar events or instructions.

Some state-space-based metrics can also be considered functionality-based metrics: Each state in the abstract machine specifies a snapshot task, and each path requiring coverage constitutes a temporal task. The projection of the circuit behavior to the state machine defines the coverage tasks.

For a small minority of designs, such as those that conform to standard interfaces like bus protocols, commercial test suites are available. For the remaining majority, however, the designer must thoroughly understand the design to define effective functionality-based metrics.[3] Useful metrics are the fruit of accumulated experience with a family of designs. As such, metrics referring to functionality fragments are highly design dependent, and defining and implementing them requires significant manual effort and expertise. This is especially true for parameterized designs that can have many configurations.[5,19]

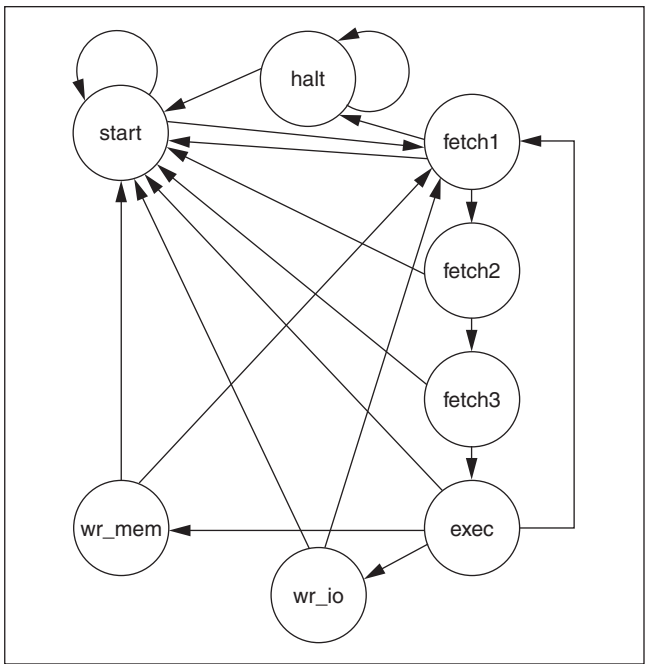

Figure 2. The control finite state machine for the Viper microprocessor. Source: *Journal of Electronic Testing: Theory and Application*.[13] Reprinted with permission.

Test generation targeting functional metrics is also more complex than test generation for structural coverage metrics. Because the metrics' constraints span several clock cycles and are not necessarily localized in HDL code or the circuit net list, automated test generation is difficult. The difficulty is compounded if a coverage task involves a certain interleaving of, or timing constraint between, events from different modules or external sources. Although this approach catches more of the difficult-to-find bugs than most other approaches, the effort involved in generating tests is daunting. Therefore, directed tests guided by functionality coverage metrics consume only a minority of simulation cycles.

An alternative approach to deriving functional coverage tasks is to specify them while developing the design's test plan. Functional coverage in this case involves a self-checking test suite that monitors whether the system is behaving as intended.[4,5] However, self-checking tests are not well-suited for directing test generation toward unexpected corner cases (rarely exercised scenarios that may escape the designer's attention).

Because they focus on the probable sources of error as determined by either experience or intuition, functionality-based metrics constitute the most direct attempt at tailoring coverage to particular types of bugs. Unlike other metrics, complete coverage of functional tasks provides a list of cases guaranteeing the design's behavior. For complex systems, a suite of functional coverage metrics is indispensable to the validation process, and this valuable intellectual property can be transferred onto newer-generation designs.

Other emerging metrics

Here we discuss three important attributes of coverage metrics that are not fundamentally tied to the description format on which they are defined: error models, observability, and applicability to specifications.

Error (fault) models

Each coverage metric has an underlying error, or fault, model. In some instances, this connection is obvious, such as with toggle-coverage or stuck-at faults, assertions, and assertion violations. In other cases, the error model is not made explicit, and only a loose connection exists between the metric and the errors it is intended to catch. For instance, a coverage metric requiring that certain instruction sequences be exercised may be intended to catch all erroneous behavior of a pipeline's interlock logic.

Some coverage metrics are defined directly by reference to an error model. These fault-based metrics mimic mutation coverage for software testing and manufacturing testing of hardware. Fault-based metrics model a hypothetical design error by a small local mutation in a design description format such as a net list, HDL code fragment, or state transition diagram. A simulation run has detected a fault if that single fault causes the design to behave differently than it would without the fault. Fault coverage, therefore, is the percentage of faults that the test suite would have detected.

Examples of fault models include gate (or input) omission, insertion, and substitution;[20] modification of outputs or next states for a given input;[21] minor modifications to HDL code;[22] and bus single-stuck line errors.[23] The major weakness of these metrics is that the fault models are motivated more by ease of definition and use than actual correspondence to design errors. Moreover, ease of measurement usually necessitates additional restrictions, such as the single-fault assumption[22] or the assumption that faults are not masked later in the simulation run.[21] The connection with design errors is made either intuitively[21] or on the basis of limited experimental data.[22] Despite these shortcomings, design fault coverage is instrumental because it provides a fine-grained measure of how adequately a design is exercised. Although each design fault error model may be simplistic and local in nature, exploring the design's behavior in sufficient detail to detect all hypothetical errors can help discover actual design errors.

Observability

In a simulation-based validation framework, the system's implementation-level model runs in parallel with a reference model described at a different abstraction level or with monitors and assertions that check for certain behavior in the implementation.[5] The implementation model's variables that are checked against required behavior in this manner are *observed* variables. Although the remaining variables' values are easily visible during simulation, correct behavior is not specified, and determining by inspection whether these variables are behaving correctly is difficult. Hence, they are called *unobserved* variables. A discrepancy from desired behavior is detected during simulation only if an observed variable takes on a value that conflicts with the value specified by the reference model. Therefore, a unit of a design's structure or functionality delineated by a metric should be considered covered if and only if it is exercised and a hypothetical dis-

crepancy originating in that unit causes an incorrect value in an observed variable.

The observability requirement does not mean that a portion of the design must behave incorrectly during simulation or that the design must be simulated with an artificially injected error. The only requirement is that the portion must affect the value of an observable variable in some clock cycle. Thus, designers are assured that had there been an error in that portion, the framework would have detected it. It is also important to distinguish between unobservable and unobserved errors. An unobservable error is one that no simulation run can detect. If all variables that are crucial to the design's functionality are observable, then in the worst case, unobservable errors will cause performance penalties but not incorrect behavior. An unobserved error, however, could propagate to an observable variable during a simulation run. The fact that we have excited but not observed the error merely indicates that we have not driven the design with the appropriate inputs to propagate the error to an observed variable. Not taking observability into account can result in an artificially high reading of coverage and a false sense of security. In an extreme case, designers could end up exercising entire design portions without checking their behavior. Consider the following example:

```
i = j + k;
x = c * i;
if (a > 0)
  o = x;
else
  o = 0;
```

Suppose the circuit portion that computes `i` has an error. If the inputs to the simulator are such that whenever `i` has an error, `c = 0` or `a < 0`, and `o` is the only variable that gets compared with the reference model during simulation, the error in `i` will never be detected.

Few coverage metrics make explicit reference to the notion of observability. Noteworthy exceptions include a state-space-based metric,[21] an observability-based code coverage metric,[22,24] and an observed-variable-based metric.[25] The state-space-based metric[22] has requirements to ensure that a transition tour of an abstract state machine uncovers all output and transition errors in the implementation. The requirements specify what part of the state must be visible to the reference model. With the observability-based code coverage metric,[22,24] each assignment is tagged in HDL code, and each tag must propagate to an observed variable during simulation. Thus, an observability requirement augments the code coverage metric. The observed-variable-based metric[25] evaluates the completeness of a set of formally verified properties in terms of an observed variable. Even when a design's state space is searched exhaustively, the tested properties' comprehensiveness must be quantified. This scheme considers a state covered if changing the value of an observed variable affects the truth of one of the properties being checked. In all three examples, evidence shows that a metric integrating some notion of observability is a superior measure of validation quality.[22,24,25]

Metrics applied to specifications

Following the convention in software testing literature, we call metrics defined on the description of a systems' implementation *model-based metrics*. Such metrics are weak at detecting missing functionality. Because a design's specification encapsulates the required functionality, metrics defined on specifications are better suited for this purpose. Assuming formal specifications exist, it makes sense to apply or generalize model-based coverage metrics to specifications as well.[3] For example, obtaining full coverage for executable specifications must be a minimum requirement. If the specification is in the form of a collection of FSMs, it is reasonable to apply, in increasing order of sophistication, metrics that are defined on FSMs. In addition, some software coverage metrics that are difficult to apply at the implementation level, such as domain or mutation coverage, may be applied to specifications.[3]

Ideally, a specification should encapsulate only the design functionality and none of the implementation details. In this way, the specification gives a checklist of exercisable behaviors obtained independently from the design structure. For certain design families, measur-

ing simulation coverage on various forms of specifications is common. For others, no formal, executable specifications exist, and specification coverage is measured informally. Formal specifications let verification engineers automatically measure coverage during each simulation and obtain statistics about various aspects of the specification.

Specification-based metrics alone may not exercise a design thoroughly. Because the specification is not aware of implementation choices and optimizations, two simulation runs that look equivalent according to the specification may exercise different portions of the implementation. Therefore, model- and specification-based metrics should complement each other.

FUNCTIONAL VALIDATION of hardware designs relies heavily on software simulation. A suite of coverage metrics is vital for the success of simulation-based validation. Even when formal, exhaustive methods are applied to larger design portions, coverage metrics are necessary to check the comprehensiveness of the properties and quantify the validation effort in general.

For most reasonable definitions of coverage metrics, the effort invested in deriving the metrics and measuring them pays off in the form of better error detection. The value of increased confidence in the design's correctness almost always outweighs the overhead of measuring coverage. Because this confidence depends on the connection between bug classes and coverage metrics, this connection must be established more concretely, through extensive studies correlating the two. This is the area where future research is most needed. Another research direction for which there is great practical demand is providing automation for functionality-based metrics. In addition to tools that aid input generation, an investigation of metrics characteristics that simplify input generation would also be beneficial. ■

Acknowledgments

We thank Laurent Arditi (Texas Instruments), Lisa Noack (Compaq), Carl Pixley (Motorola), and Moshe Sananes (Intel) for sharing their experience with coverage metrics.

References

1. C.N. Ip, "Simulation Coverage Enhancement Using Test Stimulus Transformation," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design, Digest of Technical Papers*, IEEE Press, Piscataway, N.J., 2000, pp. 127-133.
2. P.-H. Ho et al., "Smart Simulation Using Collaborative Formal and Simulation Engines," *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design, Digest of Technical Papers*, IEEE Press, Piscataway, N.J., 2000, pp. 120-126.
3. L. Fournier, A. Koyfman, and M. Levinger, "Developing an Architecture Validation Suite: Application to the PowerPC Architecture," *Proc. 36th Design Automation Conf.*, ACM Press, New York, 1999, pp. 189-194.
4. S. Taylor et al., "Functional Verification of a Multiple-Issue, Out-of-Order, Superscalar Alpha Processor—the DEC Alpha 21264 Microprocessor," *Proc. 35th Design Automation Conf.*, ACM Press, New York, 1998, pp. 638-643.
5. M. Kantrowitz and L.M. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor," *Proc. 33rd Design Automation Conf.*, ACM Press, New York, 1996, pp. 325-330.
6. H. Zhu, P.V. Hall, and J.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, no. 4, Dec. 1997, pp. 366-427.
7. Y. Malka and A. Ziv, "Design Reliability-Estimation through Statistical Analysis of Bug Discovery Data," *Proc. 35th Design Automation Conf.*, ACM Press, New York, 1998, pp. 644-649.
8. L. Arditi and G. Clave, "A Semi-formal Methodology for the Functional Validation of an Industrial DSP System," *Proc. IEEE Int'l Symp. Circuits and Systems*, IEEE Press, Piscataway, N.J., 2000, pp. 205-208.
9. R. Grinwald et al., "User Defined Coverage—A Tool Supported Methodology for Design Verification," *Proc. 35th Design Automation Conf.*, ACM Press, New York, 1998, pp. 158-163.
10. R. Vemuri and R. Kalyanaraman, "Generation of Design Verification Tests from Behavioral VHDL Programs Using Path Enumeration and Constraint Programming," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 2, June 1995, pp. 201-214.
11. R.C. Ho and M.A. Horowitz, "Validation Coverage Analysis for Complex Digital Designs," *Proc. Int'l*

Conf. Computer-Aided Design, ACM Press, New York, 1996, pp. 322-325.
12. M. Benjamin et al., "A Study in Coverage-Driven Test Generation," *Proc. 36th Design Automation Conf.*, ACM Press, New York, 1999, pp. 970-975.
13. J. Shen and J.A. Abraham, "An RTL Abstraction Technique for Processor Microarchitecture Validation and Test Generation," *J. Electronic Testing: Theory and Application*, vol. 16, nos. 1-2, Feb. 1999, pp. 67-81.
14. J. Shen and J.A. Abraham, "Verification of Processor Microarchitectures," *Proc. 17th IEEE VLSI Test Symp.*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 189-194.
15. D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "A Unified Framework for Design Validation and Manufacturing Test," *Proc. Int'l Test Conf.*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 875-884.
16. S. Ur and Y. Yadin, "Micro Architecture Coverage Directed Generation of Test Program," *Proc. 36th Design Automation Conf.*, ACM Press, New York, 1999, pp. 175-180.
17. D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," *IEEE Trans. Computers*, vol. 47, no. 1, Jan. 1998, pp. 2-13.
18. S. Ur and A. Ziv, "Off-the-Shelf vs. Custom Made Coverage Models, Which Is the One for You?" *Proc. Software Testing, Analysis, and Review* (STAR 98), CD-ROM, Software Quality Engineering, Orange Park, Fla., 1998.
19. M. Puig-Medina, G. Ezer, and P. Konas, "Verification of Configurable Processor Cores," *Proc. 37th Design Automation Conf.*, ACM Press, New York, 2000, pp. 426-431.
20. D.V. Campenhout et al., "High-Level Design Verification of Microprocessors Via Error Modeling," *ACM Trans. Design Automation of Electronic Systems*, vol. 3, no. 4, Oct. 1998, pp. 581-599.
21. A. Gupta, S. Malik, and P. Ashar, "Toward Formalizing a Validation Methodology Using Simulation Coverage," *Proc. 34th Design Automation Conf.*, ACM Press, New York, 1997, pp. 740-745.
22. S. Devadas, A. Ghosh, and K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," *Proc. 33rd Design Automation Conf.*, ACM Press, New York, 1996, pp. 418-425.
23. D.V. Campenhout, T. Mudge, and J.P Hayes, "High-Level Test Generation for Design Verification of Pipelined Microprocessors," *Proc. 36th Design Automation Conf.*, ACM Press, New York, 1999, pp. 185-188.
24. F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Simulation," *Proc. 35th Design Automation Conf.*, ACM Press, New York, 1998, pp. 152-157.
25. Y. Hoskote et al., "Coverage Estimation for Symbolic Model Checking," *Proc. 36th Design Automation Conf.*, ACM Press, New York, 1999, pp. 300-305.

Serdar Tasiran is a research scientist at Compaq Systems Research Center. His research interests span all areas of computer-aided design—particularly validation, formal verification, and logic synthesis. His most recent work focuses on coverage-directed validation of microprocessors. Tasiran has a BS in electrical engineering from Bilkent University; and an MS and PhD, both in electrical engineering and computer sciences from the University of California, Berkeley. He is a member of the IEEE.

Kurt Keutzer is a professor of electrical engineering and computer science at the University of California, Berkeley, where he also serves as associate director of the Gigascale Silicon Research Center. His research interests span a wide number of areas in computer-aided design. Keutzer has a BS in mathematics from Maharishi International University, and an MS and PhD in computer science from Indiana University. He is a Fellow of the IEEE.

■ Direct questions or comments about this article to Serdar Tasiran, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301; serdar.tasiran@compaq.com.