Metrics in SoC Verification

Not just for coverage anymore

Andreas Meyer and Harry Foster
Design and Verification Technology Division
Mentor Graphics Corporation
andy_meyer@mentor.com, harry_foster@mentor.com

Abstract—**Process metrics provide a clear, quantitative and objective measure to assess process performance and progress towards a specific process goal. SoC functional verification involves integrating multiple IP blocks. So understanding how to define, measure, correlate, and analyze appropriate IP and system-level metrics is fundamental to improving performance and achieving quality goals. Yet, many of today's SoC project members' understanding of metrics is often limited to simple coverage measurements. In this paper, we take a broader view of metrics—beyond traditional coverage measurements—and identify a range of metrics across multiple aspects of today's SoC functional verification process. We then discuss other important considerations when integrating metrics into a project flow, such as metric categorization, run-time control, data management, and reporting and analysis.**

Keywords-functional verification; coverage; metrics; SoC; IP

1. INTRODUCTION

Metrics provide a way to build measurements into a design or verification process and environment in such a way that specific issues of interest can be monitored, and then corrective action can be taken when problems are identified.

With any complex design, no single measurement will give an accurate portrayal of a project's state. Each measurement can only give one view and most have significant limitations, which is why a wide range of metrics are often used to build a reasonably accurate picture of a project. With many different types of metrics, good planning, and solid analysis, it is possible to build a view of the project that is not distorted by the drawbacks of any one measurement. Yet because metrics can be expensive to implement and maintain, planning is also critical to gaining a meaningful view of the system in a cost-effective manner.

1.1 Paper scope

The focus of this paper is the philosophy that underpins creation of a metrics-driven SoC Verification process. We do not discuss the actual implementation of the metrics process since the implementation details would be project specific. Furthermore, we do not discuss the actual tools used to implement a metrics process. We believe that it is first necessary to understand *what* is required of a metrics-driven SoC verification process before delving into the details of *how* to implement the solution.

1.2 Prior work

Applying metrics to quantitatively improve a process is a fundamental component within the Capability Maturity Model (CMM), a framework for assessing and improving software processes originally developed by Carnegie Mellon University and the Software Engineering Institute. [1] For hardware verification, coverage is one metric that has been used for years. The book *Functional Verification Coverage Measurement and Analysis* [2] provides an excellent overview and taxonomy of various coverage measurements. In addition, the book *Metric Driven Design Verification* [3] provides an introduction to metrics-driven processes in hardware design and verification. What differentiates this paper from prior work is that we focus the metrics-driven discussion on issues, challenges, and concerns specifically related to SoC design and verification.

1.3 Paper organization

This paper is organized as follows—In Section 2, we describe the forces driving change in today's SoC verification flow. In Section 3, we discuss what can be measured in an SoC verification flow and how these measurements can be used. Section 4, the bulk of the paper, describes important considerations when architecting a metrics-driven process. Although this paper does not focus on the implementation details, in Section 5 we do discuss important considerations during process implementation. Finally, Section 6 provides some concluding thoughts on what to expect after adopting and implementing a metrics-driven process.

2. WHAT ARE THE DRIVING FORCES FOR CHANGE?

In this section, we begin by examining the issues that are motivating change and the need for metrics-driven processes. We then discuss what is not working in today's IP-based SoC design flows.

2.1 How is IP-based design changing?

The increasing number and complexity of IP blocks being integrated into a single chip is driving the need for process metrics. In the past, when IP blocks operated independently of each other, SoC verification consisted mostly of checking the interconnects of each IP block and the registers across the SoC, as illustrated in Figure 1.

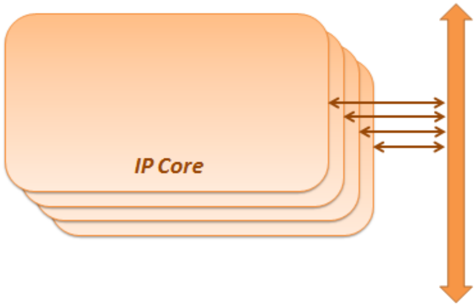


Figure 1. Yesterday's concern—basic IP interconnects

This level of verification doesn't require an understanding of IP blocks' internals. Metrics could provide some utilization information (e.g, Answering questions such as: What is the utilization on a particular bus? Are we getting access to memory fast enough?) which are important for performance analysis but not likely to be critical in terms of identifying errors that result from interacting IP blocks.

Today, as IP blocks interact directly with each other, it is critical to verify functionality between IPs at the SoC integration level. Even with IP blocks that operate independently, shared resources mean that the behaviors of one block can affect other blocks. As part of the integration effort, bus utilization, fairness and memory sharing may need to be examined to determine whether the SoC functions as specified.

What is emerging are more complex IPs where state may be shared across multiple IP blocks, which means that functionality can only be fully tested at integration. Figure 2 gives an example of this emerging challenge, depicting multiple complex IP blocks containing a coherent cache and a memory subsystem.

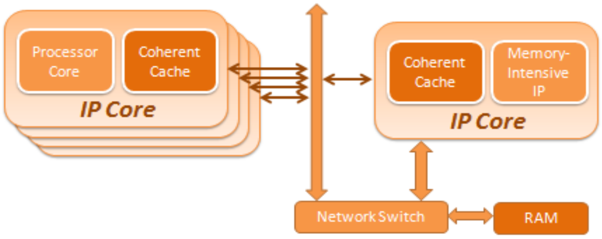


Figure 2. Tomorrow's concern—shared state

The integration of complex IP brings a new set of challenges to the SoC verification team. Each IP block is generally developed by its own team focusing on that specific block. When the blocks are integrated, the SoC verification team must debug and analyze the system, often without the knowledge of the IP internals. Each IP is a black-box to this team, which will not have the time or background to understand the inner workings of multiple complex IPs.

This requirement – productively verifying complex interactions of a full system without understanding the component parts – is driving change in how metrics are used. Without metrics, determining what happened in a simulation has become very difficult for SoC designs. Specifically, in large environments, what is not measured is not known.

2.2 What is not working?

Coverage measurements are probably the best-known metric for measuring whether a specific feature or function has been exercised by a verification test. While this is a useful thing to know, in today's environments it is a limited bit of information. Moving beyond traditional coverage, we often need additional insight into what is being verified. Examples include:

- What abstraction level was the IP (i.e., feature) instantiated at when it was covered?
- What was the integration level of the environment when closing various coverage items, (e.g., block- or system-level)?
- What stimulus was used to reach a covered item?

The increasing number and types of IP blocks being integrated into a single environment has brought new challenges in understanding what SoC functionality has been verified. A few examples related to IP simulated activity include:

- Were the complex programming requirements for a particular IP block verified?
- Were the various IP block power management features properly verified?
- Were the complex system interactions between multiple IP blocks verified?

As IP blocks become more diverse with firmware, multiple abstraction levels, and configuration options, additional metrics beyond coverage are required, metrics the help answer questions such as:

- Which IP blocks (and versions) were included in the build process?
- What firmware version was used during simulation?

Finally, moving beyond traditional coverage metrics bring up new issues in storage and ability to query of information. We explore this topic further in Section 5.

3. WHAT CAN METRICS TELL US?

We are interested in a broad set of metrics that cover the entire verification flow, giving insight into the build, simulation and regression processes—as well as various aspects of the overall project. Yet we are also interested in metrics that are actionable lest the process of measuring and storing metrics data waste project resources.

In Table 1, we provide a typical set of processes and focused areas associated with a general SoC verification flow, along with a list of process attributes that we might choose to track using metrics. In general, a single metric associated with any particular attribute in the table is of little use. Only when multiple metrics are correlated during analysis does real value emerge.

For example, tracking coverage trends over time might be interesting, though simple coverage metrics generally do not provide the insight necessary for understanding what has been verified in complex SoC designs. More complex questions associated with the SoC verification process must be answered. Continuing with our example, we might be in a situation where we built a new revision of the system, and we would not only like to know what coverage was hit, but also to examine this in the context of specific applied stimulus—say, for a particular level of abstraction of the design and for a particular revision of the firmware. Within this context, and by correlating multiple metrics, we have a clearer view of the circumstances that allowed us to hit specific coverage and the outstanding problem areas.

We now expand our discussion on *what metrics can tell us* by providing examples for various common processes within today's SoC verification flow.

3.1 Metrics as part of the build process

The build process instantiates multiple IP and testbench blocks to form a system to be verified. At this point, when appropriate metrics are defined and implemented, information on the actual build process can be obtained. Such information is most likely to be useful in large SoC environments with significant code churn. For example, we might be interested in knowing exactly which IP blocks were used during the build process, where each IP block originated, which version number was associated with each IP block and what level of abstraction was used for the build. An important point: metrics need not count multiple events to be useful. In our example, metrics for the non-event-based build can be used to qualify queries around specific code. For instance, we could determine the coverage metrics associated with a specific IP version. Correlating event-based with non-event-based metrics may be useful in checking completeness of overall verification.

Other relevant information may include configuration or randomization that was done during the build. Gathering these sorts of metrics on these aspects of the build process should be sufficient to understand what occurred or was accomplished in the build process so that errors or progress information taken from a regression can be correlated to specific components used within simulation.

3.2 Metrics as part of the simulation process

The majority of our process measurements are likely to occur during testing within each simulation or emulation run. There are a number of basic areas where reports can be useful to determine what happened during simulation, which pieces of the simulation environment were used, and how the pieces played together.

The follow are various aspects of the simulation process where measurements can be useful.

Stimulus Sources: Larger systems are likely to use a number of stimulus sources within a single simulation, which is likely to include at least one test and also include other sources, such as noise generators, software running on an embedded processor or connections to external processes. Metrics can measure which sources were used and provide information about the type and frequency of traffic generated by each source. This information can be helpful to understand how the system has been tested and to measure the productivity of various stimulus methods.

Checking Methods: As with stimulus sources, most projects are likely to use many different checking methods. Metrics can be used to ensure that the desired checkers are in place and receiving traffic to check. Beyond that, metrics can identify the numbers and types of checks that were performed, which can give an indication of how the system is being tested and how well it is performing. Measuring that the desired checkers are in place, as well as the number and types of checks used, can provide an indication of stimulus coverage and system metrics, such as traffic density, bus utilization, or system-specific operations. Checking metrics, when correlated with other metrics, may help provide a deeper understanding of the environment's effectiveness and productivity.

TABLE I. VARIOUS VERIFICATION PROCESS METRICS

Process and Focused Areas	Process and Focused Area Attributes and the Information Associated Metrics Can Provide			
Design	Abstraction level	Simulated performance	List of instantiated blocks (and versions)	
Stimulus	Source of stimulus	Type of stimulus (CR, firmware, graph, legacy, etc.)		
Checking	Source of checkers	Results of checkers	Checker abstraction levels	
IP	Interface activity	Key internal states		
Coverage	Categories of coverage	RTL/stimulus/checker reference model	Abstraction level of coverage	
Build	Source and rev of files	Initial configuration used		
Run	Simulator/Emulator	Host machine info (memory, disk image distance, etc.)	Simulation performance	Revision of tools
Debugging	Area of failure	Commonality of cases where many tests report same failure		
Regress	Which simulations	Errors found	Errors re-found (i.e., wasted simulation)	Improvements in coverage results
Bug Status	Open bugs	Bug discovery info: stimulus, abstraction level, checker…	Metrics used to isolate bug	Bug closure information (sim time, engineer time, number of runs)

Coverage Metrics: These metrics are most commonly associated with simulation. Code coverage methods require no additional coding of metrics since this information is extracted by the simulator and is useful for identifying code that has never been exercised. Functional coverage metrics need to be architected, planned, written and maintained, but they can also provide domain-specific information on the reach of the stimulus within the simulation as shown in Figure 3. Both metrics are useful for identifying holes in the input stimulus for activating lines of code, structures or behaviors within a design. Yet, by themselves these metrics cannot answer the question: “Did a specific test both activate and then propagate an event of interest to a specific checker?”

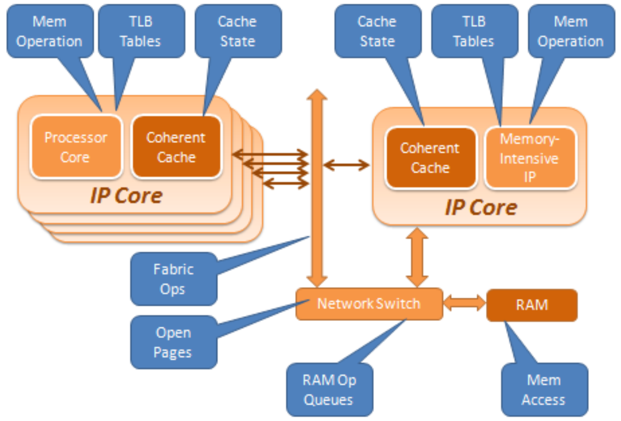


Figure 3. Domain-specific metrics

Domain-Specific Performance: In some systems, understanding the performance characteristics of the system under test is a critical part of functional verification; metrics can provide some data to estimate the system performance. By attaching to existing bus monitors or checkers and using transaction tracking, it may be possible to extract throughput, utilization and latency information from an existing verification infrastructure. For example, the metrics captured in Figure 3 could be used to measure domain-specific performance where discrete pieces of information are often sufficient to calculate system-performance.

Simulation Performance: Simulation can be time-intensive in large verification environments. Accordingly, simulation performance is important to ensure that regressions are run with sufficient regularity and to keep bug turnaround times as short as possible. Determining how to improve simulation performance can be exceedingly complex, and while metrics are not likely to help with that, they can be used to track performance and provide an early detection mechanism if design or testbench code has been introduced into the simulation process that negatively impacts performance. By correlating performance with code revisions, it may be possible to link performance degradations (or improvements) to the introduction of specific blocks of code.

Simulation Configuration: Each simulation may have run-specific configurations that can affect other aspects of the verification process. Probably the most obvious configuration is the random seed that was generated for each run. That seed or other parameters may be used to select tests or change the configuration of the simulation. Reporting each configuration option as a part of metrics allows configuration changes to be correlated with other metrics such as coverage, simulation performance and bug statistics.

3.3 Metrics as part of the regression process

Most simulations are run during the regression process. While the stimulus and coverage metrics should be reported from within each simulation, the regression process is also responsible for a number of decisions. Metrics can help ensure that regressions are efficient and productive. There are usually two types of metrics tracked in the regression process: information on the regression run and information on the simulation farm.

Regression run information may include test names, frequency of tests, random seeds, configuration choices and so on. This information, in addition to coverage, bug status and other simulation metrics, can provide insight into the effectiveness of tests and where additional tests are needed. One well-known example of this is test ranking. By looking at the tests that provide the most coverage or are most effective at uncovering bugs, test run order or test frequency can be adjusted to increase verification productivity.

3.4 Metrics as part of the overall project

There are a number of areas within a project where metrics can provide insight into various aspects of a project, even though they are not actually within a simulation. One obvious measurement is in bug reporting. Understanding bug status can provide insight into the simulation, testing, coverage and overall progress. Ideally, when a bug is detected, knowing which simulation reported the bug provides information about the stimulus, checking, versions and abstraction level used to find the bug. Knowing more details about the bug (e.g., which block, type of bug and so on) can provide information about test effectiveness and even coverage. Knowing when the bug was closed can provide information about the project progress and insight into when a particular test does not need to be run in regressions.

Other metrics outside of simulations can be useful when measuring project-wide progress. For example, how often code is checked into the revision control system is related to stability and maturity of the design and testbench. Team status metrics are also useful, particularly when multiple geographic locations are involved. Determining which metrics are useful is likely to depend on many aspects of the project, the expected lifespan of IP that is being developed and corporate culture.

4. What Is Needed To Address The Problem?

This section discusses four important aspects of a successful metrics-driven process: understanding the landscape, categorization, run-time control and reporting.

4.1 Understanding the landscape

Successful adoption of a metrics-driven process requires first recognizing the potential breadth of metrics and the imperative to organize the execution, classification and reporting of metrics during the project planning phase.

4.1.1 *Breadth of metrics*

The higher the number of metrics, the more important it is to run and report these metrics in a structured fashion. The sheer volume of metrics can easily overwhelm both the simulator and the user. Managing this issue means architecting the metrics-driven process for easy control of the execution, classification and reporting of metrics—thus permitting the user to focus the measurement on relevant areas of concern while minimizing noise produced by non-relevant metrics.

Metrics must be designed into the process in a way that can be managed and understood without detailed knowledge of each design block. That requires architecting a metrics solution that leverages the concepts of *modularity* and *APIs*, which allow the metrics to be controlled.

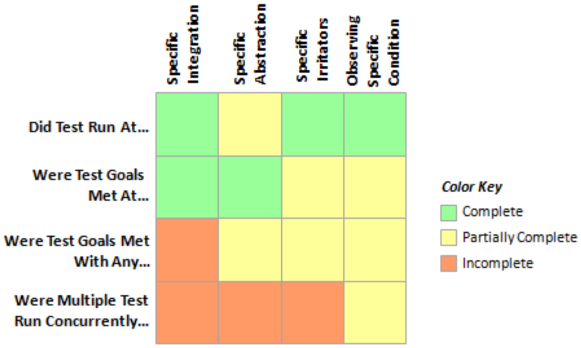
Designing the metrics solution in from the bottom up means that metrics are written by the engineers that understand the block for use with block-level verification. With well-defined APIs, that block-level metrics solution can then be integrated into a higher-level subsystem and system verification environment.

4.1.2 *Organizing metrics*

One method of managing the breadth and volume of metrics is to initially organize them into high-level areas of focus and then provide controls enabling measurement of relevant areas of interest. Three ways in which metrics can be organized are test-specific, user-specific, and project-specific organization of metrics.

Test-Specific Organization of Metrics: The relevance of a given metric changes along with the execution of various simulations, each of which generally focuses on a specific design area. Some simulations, for example, may run stimulus focused on specific system components, while others may spread activity across a broad set of components. When specific components are targeted, then the associated metrics (which are likely IP-specific, stimulus, checking and simulation) are usually relevant. When simulation activity is spread across an entire SoC, low-level metrics within specific components are likely to be less relevant, while higher-level metrics (perhaps measuring bus and API activity) may be more relevant. The key point is that these factors should be considered as you architect your metrics-driven process.

Figure 4 illustrates a conceptual checklist matrix that can be used to measure completeness of a specific test. The matrix provides the user with a method of answering the questions: Did a specific simulation run at the appropriate levels of abstraction? Were multiple tests run concurrently with a specific irritator? Obviously, the conditions measured within an actual checklist matrix would be design-specific, yet perhaps this simple example hints at the power of graphical analysis of metrics.



	Specific Integration	Specific Abstraction	Specific Irritators	Observing Specific Condition
Did Test Run At...	🟩	🟨	🟩	🟩
Were Test Goals Met At...	🟩	🟩	🟨	🟨
Were Test Goals Met With Any...	🟧	🟨	🟨	🟨
Were Multiple Test Run Concurrently...	🟧	🟧	🟧	🟨

Color Key
- 🟩 Complete
- 🟨 Partially Complete
- 🟧 Incomplete

Figure 4. Checklist matrix for test-specific analysis

User-Specific Organization of Metrics: A user may want to change the relevance of metrics at different points within a project. This decision will depend on the issues that the user wants to focus on. For example, enabling metrics in specific areas can augment information provided by traditional checkers and monitors. In addition, the user will want to enable metrics that focus on understanding how the environment was constructed and initialized and how it is running.

Project-Specific Organization of Metrics: A number of metrics may be used at the project level to measure productivity and progress. These metrics may include simulation time, build information, farm execution and broad measurements of the environment (e.g., system under test stimulus, checkers and abstraction). Such measurements are generally relevant for all simulations and are likely to be enabled on all runs as a way of capturing and monitoring the overall project.

One example of a project-specific metric might be tracking how long a regression simulation takes per various IP blocks. For example, Figure 5 shows the regression run for the Coherent Cache IP block previously illustrated in Figure 2. Notice the sudden spike in regression time on week 17. This might be caused by design or coding issues associated with a recent modification to this particular IP block. You can see here that metrics in this case allow us to respond to issues before they get out of hand.

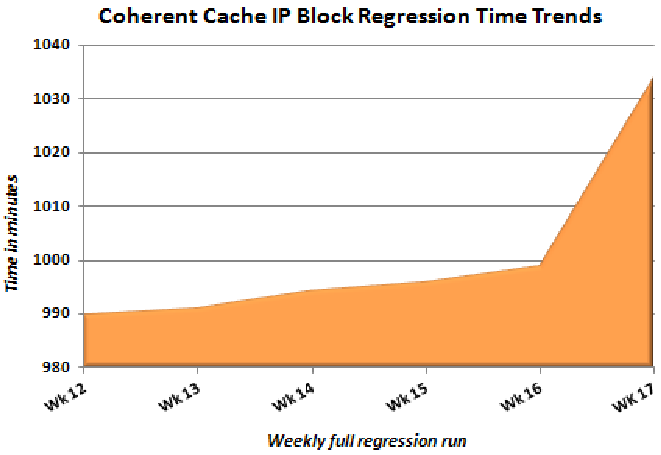


Figure 5. Simulation regression time for a particular IP block

4.1.3 Categorization of metrics

The organization of metrics discussed in the previous section proposed a more general, high-level approach to viewing and managing metrics. An orthogonal way to view metrics is to create a more focused approach by grouping measurements through categorization. For example, in system simulation, an integrated IP block may be measuring some aspect of low-level functional coverage, which is only expected to be covered in an IP-level simulation. Categorization can be used to identify which metrics are likely to be of interest in which cases and to disable metrics that are not of interest for a specific class of simulations.

The following are a few examples areas where categorization can be used to improve the performance of a metrics-driven process.

Allow specific concerns to be addressed: Complex system verification environments often lead to use of various classes of simulations. One example is the choice of abstraction level. Higher-abstraction-level simulations are generally used to allow for faster simulations with less accuracy. This approach can be useful for testing some higher-level concepts or performing long simulation runs involving firmware. However, the reduction in accuracy may require disabling a whole group of metrics. Once again, categorization can be used to specify a group of metrics that should be disabled during simulation.

Improve regression efficiency: If most regression runs pass with only occasional failures, then it may be useful to have the regression environment run with most or all metrics disabled first, and then decide whether to rerun for failing regressions. If failure types are classified, then the regression environment may be able to look up the desired metrics categories for the particular failure type and rerun the test automatically. This approach may provide a reasonable tradeoff between regression efficiency and the engineering need to debug the failure.

Allow a team to package information with an IP through categorization: When an IP is packaged and delivered for system integration, the developers have IP-specific knowledge that they may want to package up for use at the system level. Metrics categorization can help with this goal by allowing the IP designers to correlate metrics to system-level requirements.

For example, the IP designers might add a category of functional coverage to the packaged IP that they consider important at the system level. This category is likely a subset of the full functional coverage metrics. Similarly, IP-specific performance metrics are not likely to be of general use at the system level, and these metrics could be disabled through appropriate categorization. Furthermore, a category of metrics might be used to control how multiple IP blocks behave at various integration or abstraction levels.

Category definitions are likely to depend on the requirements of the specific project and company. Standardizing the category definitions and implementation across projects improves the ability to use the categories as IPs migrate from project to project. Some general categories make sense for a wide variety of projects, though it's also useful to define project- or IP-specific categories.

4.2 Runtime control

For a simulation run, the goals of each particular simulation are generally understood. For example, some simulation runs focus on specific areas of a design, while regressions may be used to explore the random stimulus space and check for correct behavior. Because metrics can require considerable simulation resources, it may be desirable to only have those resources enabled that help reach the goal of the simulation. When a simulation is focused on one area of the environment, it is reasonable to have all metrics from that area enabled and very few metrics from other areas. For regressions, coverage metrics may be the most important with, perhaps, some metrics for performance included as well.

For performance reasons and to reduce the likelihood of introducing a change in the simulation behavior as a result of changing a monitor, it is important to provide the user with a run-time mechanism to turn on metrics without having to recompile the design. Run-time mechanisms can be used to enable and disable categories of metrics at the start at specified time of the simulation. This can improve an environment's performance and random stability. If care is not taken, just enabling a monitor can affect a design's random stability. OVM and UVM are structured to minimize the likelihood of an instantiation change resulting in a randomization difference.

4.3 Reporting

In this section, we discuss various aspects of a metrics-driven process related to reporting, ranging from the use of metrics for trend analysis and queries to the collection and storage of metrics data.

4.3.1 Using metrics

Among the most common ways to use metrics are: trends over time and correlations across different metrics. Which metrics are used and how they are reported are likely to be project-specific. Metrics can be plotted over time to show trends, or they can be calculated as a single query to answer a specific question. In general, trend analysis is the most common use of metrics.

4.3.1.1 Trends

Plotting metrics over time is one way to determine progress and direction within a project. This approach can be particularly useful to check progress against a schedule or to determine the effectiveness of specific verification methods. The simplest reports might show a single measurement over time, for example, bug-open and bug-closure rates plotted over time or the percentage of tests written over time.

A slightly more complex report might include the correlation of multiple metrics plotted over time. Again, the idea is to choose a group of metrics that, when analyzed together, provide a useful view into the project. Code and functional coverage reports fit into this category. Coverage is generally measured as the ratio of covered to uncovered lines of code (or functional coverage points, in the case of functional coverage). By plotting that ratio over time, a trend can be seen, and hopefully, that trend shows that code coverage is increasing as time goes by.

4.3.1.2 Queries

Metrics can be used for other reasons. For example, a manager might query the database of metrics to determine performance of either the simulator or the DUT. Alternatively, a manager might query the database of metrics to determine the effectiveness of specific verification components.

The following are a few examples of queries that one might encounter on a typical SoC verification project.

Query Test-specific example: In the case of test-specific queries, a verification engineer might be interested in knowing if a group of tests designed to cause specific interactions between blocks within the DUT actually worked. To check this condition, the engineer could have created a monitor that provides feedback into the stimulus source. However, this approach might require additional verification infrastructure. Alternatively, general coverage results are not likely to be effective in determining this condition since they tend to be accumulated over multiple tests. A metrics query may be the simplest way to determine if the test achieved its goal, particularly if multiple data points are needed. As an example, a query can be used to determine if a specific test, when run at a specific integration level in which several specific blocks were instantiated, caused a specific cover point to be hit. This type of unique and specific query may well be used as a part of determining the completion status of a regression, but it is unlikely to be of interest as a general trend discussed in the previous section.

Query simulation performance example: Although simulation efficiency can be measured in many different ways, the most straightforward is as cycles per second (provided the bus frequencies within the DUT are constant). Studying cycles per second can help detect the introduction of inefficient code when you log the frequencies across IP blocks as they migrate from standalone environments into subsystem integrations. Making simulation performance a criteria for revision control system check-ins can reduce system simulation time.

In larger systems, a more interesting performance measurement may be productivity per simulation cycle, which can be captured through the number of tests, checks and cover points achieved per cycle. One could also track the number of RTL bugs caught per cycle, per category of stimulus, or per regression. These types of reports can provide insight into the effectiveness of the verification environment, and they might also be worth plotting over time to view trends.

Query architectural performance example: Tests can help confirm that specific DUT operations perform as expected and look at the performance of a group of operations. However, by definition, any test can only measure DUT performance within the confines of that test.

Metrics help make broad and specific performance measurements. Such measurements may be fairly straightforward, including bus utilization, cache hit ratios, processor operations per second or data transfer ratios. More complex calculations such as snoop filter effectiveness or quality of service measurements can also be of interest.

By tracking performance measurements and correlating them with stimulus sources or integration levels, patterns may emerge and show where improvements are needed. The main requirement is that there are reasonably consistent measurements across a range of environments. Where performance is a critical concern, plotting metrics over time can also help provide insight into the project completion criteria.

All of these query examples are based on the concept that metrics allow for capture and analysis of data across integration levels, abstraction levels, IP blocks and time. Because the metrics are stored outside of the simulation, new queries can use all the historic data to provide insight and trend information.

4.3.2 Collection and storage

Storing metrics in an addressable manner (for example, database keys representing their organization and categorization) facilitates data accessibility, permitting the data to be accessed in queries. Generally, there are two types of queries that need to be run: *predefined* and *custom-designed*. Predefined queries are used frequently. They are often accessed through a webpage by a wide group of people to understand the current and progressing state of the project. On the other hand, custom-designed queries look at a certain aspects of the project. This type of query is likely to be narrow in scope and used by a small group of engineers to understand a specific issue.

When architecting and implementing a metrics-driven process, it is important to recognize that the storage system must be able to easily accommodate both types of queries, while simultaneously accepting new metrics that are gathered from each simulation run. With a large simulation farm and continuous regressions, the collection and storage of metrics is a significant task all on its own. A single engineer can use the stored metrics to understand a single run, but storing the metrics in a project or environment-wide way allows for additional analysis.

5. WHAT IS NEEDED TO ADOPT METRICS?

Though we will not discuss the actual implementation details, there are important aspects of an implementation that should be considered when architecting a solution. Like many aspects of functional verification, a methodology is required for implementing a metrics-driven process. Since that methodology is likely to be carried forward for several project generations, it is imperative to create a flexible methodology that is likely to benefit future projects.

In the following sections we discuss, from a high level, various aspects and considerations for an implementation.

5.1 Making the metrics solution part of the architecture

As with any other part of the verification environment, effective implementation of metrics must be architected to fit into the project and environment. Several things must be considered before implementing metrics, including the types of metrics, and what they are likely to involve:

Project-independent metrics

- These can involve rules for implementation, which include the language used and mechanisms to control when they are active.
- These can involve rules for reporting, which include how metrics provide results and how reports are correlated to specifics of a particular simulation and environment.
- These can involve storage query, and reporting mechanisms.

Environment-specific metrics

- These can involve simulator performance, including runtime and environment information for a simulation.
- These can involve regression, including farm and queue information.

Project-specific metrics

- This area is architecture-specific, and it generally starts by understanding the key busses and processes in the system and how to measure their correctness, accuracy and performance.
- Bus-specific metrics are likely to include functional coverage (which operation combinations and mixes of operations have occurred) and performance (utilization, bandwidth, queuing, backpressure, latency and so forth).

5.2 Making the metrics' solution useful

Ensuring that any specific measurement will provide value is an important part of a metrics-driven process architecture. Metrics are expensive, in large part due to their associated maintenance and simulation costs. To prevent waste, the architectural phase can be used to define what measurements are needed and what each measurement will be used for. Clarity of purpose allows designers to understand where metrics are and are not needed.

One way to provide clarity and make metrics useful is to categorize metrics by purpose. That is, for any particular IP, some metrics are likely to be used internally to help the IP designers understand the operation of the block. Other metrics may be defined for external use to measure the operation of the IP when integrated in a system environment. A clear definition of purpose allows for effective categorization of metrics.

5.3 Ensuring the metrics solution is consistent

When multiple IPs are integrated into a larger system, metrics provide one view into the environment. Consistency in a metrics solution is important for both visibility and efficiency. Consistent categorization makes it easier to control reporting based on simulation goals. Providing similar levels and quantities of reports increases the likelihood that the reports can be meaningfully analyzed. If some blocks are dramatically over or under reporting, it may skew analysis or result in reports that are ignored.

Implementing metrics requires significant expertise in both coding and project management. A metrics solution implementation, while not difficult, can be time consuming in terms of development and execution. Providing a useful set of metrics that are consistent in reporting and implementation across a range of designs requires significant planning and discipline.

A consistent development style that is efficient to implement and simulate is critical. Consistency in enabling, disabling, and categorizing individual metrics provides a means to control metrics in the simulation. Consistent reporting eases the task of extracting information from various parts of the environment into a single coherent picture.

5.4 Making the metrics solution work with legacy IP

Legacy IP provides a number of challenges to verification methodology; however, from the viewpoint of metrics, legacy and third-party IP is generally not as much an issue. These IPs are often free of any metrics, which means that from an analysis viewpoint, they are simply black-boxes that don't interfere with trend analysis on the rest of the system.

Often, legacy IP is not well understood because the developers are not available to provide support. Despite this impediment, there may still be value in adding some metrics to support the environment. For example, metrics may be added to the borders of the legacy IP block to ensure correct integration and sufficient exercise of the IP. Whether or not these types of metrics are needed should be determined during the architecture phase of the project.

5.5 Planning the metrics solution

Metrics, as with any verification effort, are expensive. Planning where metrics are required and the types of measurements needed allows developers to determine where to add metrics, how they should be categorized, what reporting is needed, and the types of analysis that may be done. Metrics development, even with careful planning, is time-consuming. Furthermore, maintaining metrics as the DUT and testbench environment evolve requires expertise and diligence.

Despite the expense in developing and maintaining the measurement of metrics, they can provide an important view that allows for measurement and improvement of the overall environment.

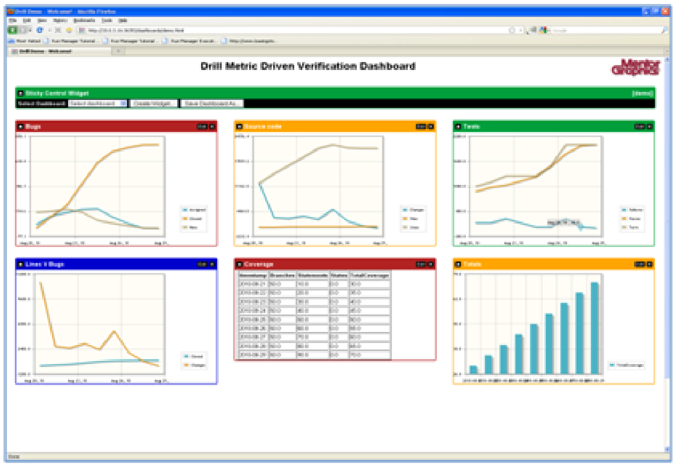


Figure 6. A project metrics-driven verification dashboard example

6. WHAT TO EXPECT AFTER ADOPTING METRICS

Metrics give a new ability to see and measure day-to-day design activity. More useful, however, are the measurements over time revealing trends and progress, as conceptually illustrated in Figure 6. The time horizon can be anywhere from days (to see the immediate progress of the design, verification environment and project) to weeks (where the project can be tracked against the schedule) to months (where teams, methodology, and tool productivity and effectiveness can be tracked, measured, and improved).

Only by knowing the current state of verification is it possible to determine what to improve or whether a change has caused the desired improvement. Gut feelings, impressions, and intuition can be effective in small projects where a few people have a good understanding of the entire project; however, in larger SoC projects that involve multiple complex IPs, no one person has a view that encompasses the entire project. As a result, the intuition of one or even several people may not accurately portray the state of the project. Metrics can be used to provide a quantitative measure of the state of a project and permit comparisons, analysis and corrections to be made.

Metrics can also be used to catch inefficiencies when they are first introduced into a system. Without these metrics, it may be weeks before a small change in an IP block causes a significant slow-down in the system environment. By then, it could be difficult to determine what change caused the deterioration.

By providing a quantitative assessment of IP quality and efficiency, metrics can track productivity by component or system and at a point in time or as a trend. This view into the system is the basis for productivity improvements and for on-the-fly detection and correction of issues with the design or verification environment.

REFERENCES

[1] Carnegie Mellon University and Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, 1995.

[2] Piziali, A., *Functional verification Coverage Measurement and Analysis*, Kluwer Academic Publishers, 2004.

[3] Carter, H., Hemmady, S., *Metric Driven Design Verification: An Engineer's and Executive's Guide to First Pass Success*, Springer, 2007.