

Assertion-Based Verification

March 2003

Introduction

The EDA industry has acknowledged that functional verification is causing a bottleneck in the design process and is inadequately equipped to deliver verification of future designs.

Today, designs can no longer be sufficiently verified by ad-hoc testing and monitoring methodologies. More and more designs incorporate complex functionalities, employ reusable design components, and fully utilize the multi-million gate counts offered by chip vendors.

To test these complex systems, too much time is spent constructing tests as design deficiencies are discovered, requiring test benches to be rewritten or modified, as the previous test bench code did not address the newly discovered complexity. This process of working through the bugs causes defects in the test benches themselves. Such difficulties occur because there is no effective way of specifying what is to be exercised and verified against the intended functionality.

This paper presents a powerful new approach to functional verification that dramatically improves the efficiency of verifying correct behavior, detecting bugs and fixing bugs throughout the design process. In this approach, the central focus of the verification process is assertions, which detect bugs, guide the test writers to write better tests and direct testbenches to produce stimuli. It raises the level of verification from RTL and signal level to a level where users can develop tests and debug their designs closer to design specifications. It encompasses and facilitates abstractions such as transactions and properties. Consequently, design functions are exercised efficiently (with minimum required time) and monitored effectively by detecting hard-to-find bugs. This methodology is called assertion-based verification, as the assertions are the primary basis for ensuring that the Design Under Test (DUT) meets the criteria of design quality. These techniques support two methods: dynamic verification using simulation, and formal or semi-formal verification using model checking and proof techniques.

Two key technologies are introduced: assertion specifications and functional coverage. Based on these technologies, a comprehensive functional verification solution is created that incorporates language, analysis tools and a methodology to produce a next-generation capability. These techniques address current needs of reducing manpower and time and the anticipated complications of designing and verifying complex systems in the future.

2.0 Improving Current Functional Verification Approaches

The current method of verifying a design involves writing tests and running simulations to exercise the tests. Monitors are spread throughout the tests, the design model, and the post-processing software or scripts. There is no concise description of the behavior that is being tested. This unorganized way creates several challenges. This section elaborates on some of the important problems and how assertion-based solution addresses them. Before we describe assertion-based verification, it is important to explain the meaning of assertion. An assertion, sometimes called a checker or monitor, is a precise description of what behavior is expected when a given input is presented to the design. To verify a design feature, first its functionality must be understood, then its behavior must be explicitly described, and finally, the conditions under which this behavior is applicable must be stated. This description is captured by an assertion. For example, one of the rules of the PCI protocol is "Once STOP# is asserted, it must remain asserted until the transaction is complete." This English language description is translated by tools to an assertion description, to ensure that the rule is exercised and is abided by whenever it is exercised. The level of assertion description can vary depending on the level at which the verification is conducted. The assertion may specify the behavior in terms of signals, transactions or higher-level design functions.

The assertion-based solution, provides a declarative language to accurately and concisely specify what is to be tested, verified and measured for quality assurance. Equally important to writing precise behavior is the ability to express behavior at a higher level than currently practiced, so that the results of the analysis done by the tools can be tied directly to the specification.

This environment is supported by sophisticated analysis tools all steered by a common assertion specification to uncover implementation problems in the DUT, pinpoint design entities that need to be fixed, and measure quality. For further productivity gains, reusable verification components are supported to ensure that the time invested in creating a verification environment can be utilized in multiple designs, and in future designs. Pre-built verification components can be plugged in to different environments that require the same verification for new design components.

2.1 Systematic Checking Ensures Quality Against Test Plan

With no concise and unified specification of assertions, it is not possible for analysis tools to automate the verification process or produce intelligent feedback to users. As a result, users are left with large amounts of data that they are required to process and analyze manually. Often, some companies invest a significant amount of verification resources to develop and maintain ad-hoc scripts to manage and post process the data, but still lack advanced analysis capabilities to achieve the productivity gains needed in the industry.

An interpretable description of assertions opens up avenues for new advanced capabilities. Tools using assertions can check the design behavior against the test plan during simulation, allow formal property checking where applicable, provide a high-level advanced debugging environment, and automatically generate tests. Figure 1 illustrates how various tools can interplay, using a common criterion of quality.

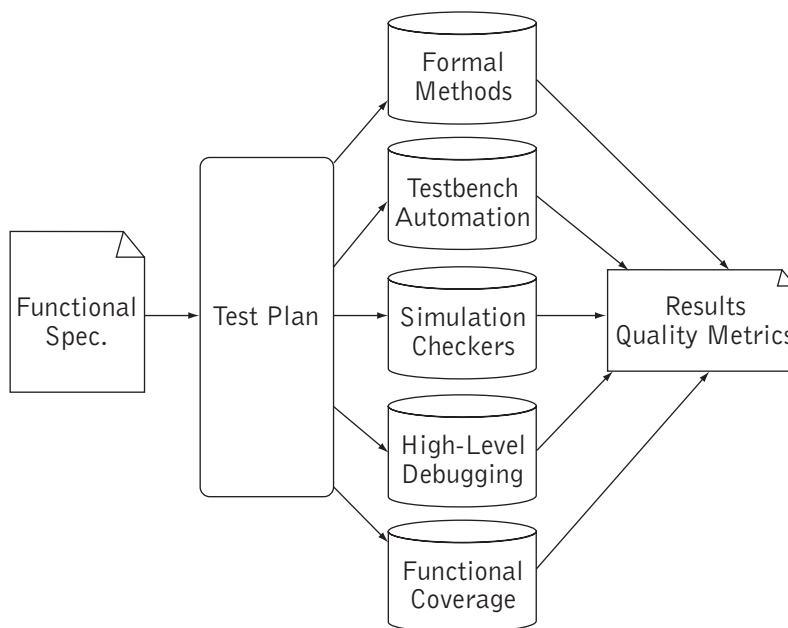


Figure 1. Systematic Approach to Quality

The verification environments shown in Figure 1 verification environment shortens the design verification time by providing automatic capabilities of correlating result data, pruning result data by removing redundant or irrelevant activities, and shield the user from having to analyze low-level data. Furthermore, intelligent tool cooperation among various design verification tasks is made possible to more efficiently and effectively probe the design activities. The outcome of the analysis consists of consolidated results and status of quality against the test plan.

2.2 Raising Level of Abstraction Simplifies Validation

Today, verification entails providing stimulus to signal interfaces of the DUT and checking the response from those signals. Typically, test plans do not detail each test in terms of signals, but rather specify a feature or a functional scenario for exercising. To allow writing checkers, debugging errors and covering functionality efficiently, the level of abstraction at which the analysis tools operate must be raised to the level at which the test plan is described.

An assertion language provides an abstraction mechanism flexible enough to suit the needs of a verification engineer, who translates a design specification to a test plan, reduces the time spent in such manual tasks, and the reduces the risk of introducing testing errors. Moreover, the supporting tools in the verification environment also provide feedback at the appropriate level as conceptualized by the engineer. This means errors are debugged faster as precise information is presented for analysis, coverage is co-related with the test plan, and more effective tests are generated to test features in terms of transactions, packets, sequences or other abstractions.

2.3 Functional Coverage Improves Quality and Efficiency of Test Benches

Another problem in conducting verification without assertion specifications is that there is no easy way to know what checks are being targeted or monitored. The tools cannot process the information produced from a simulation run to determine specific checks that have been detected as failures, exercised or re-exercised. In other words, the measurement of quality is not practical, and without this measurement, the entire design verification process suffers from a lack of coherent information about its progress.

To economize their verification effort, users critical feedback from functional coverage that directly reports the quality of the test plan. This also greatly reduces the anxiety of leaving bugs related to corner cases and scenarios unaccounted for in the functional specification of the design. Functional coverage tools take the same assertion specifications and provide a detailed account of what tests, checks, scenarios and data have been exercised.

Without access to a comprehensive functional coverage metric, random test generation has to be manually adjusted for each simulation run. The input constraints and seeds are modified so that more scenarios can be exercised to catch bugs. This process of manual trial and error is inherently inefficient and incomplete. But, the availability of functional coverage during run-time can change this situation remarkably. The coverage feedback obtained from coverage analysis drives test benches to make necessary adjustments as soon as the coverage is determined to be stalled. This process, known as reactive testbench, generates more effective stimuli for the DUT. Previous attempts at implementing a reactive testbench approach were not successful, as they were based on weaker measures of quality such as statement coverage or toggle coverage.

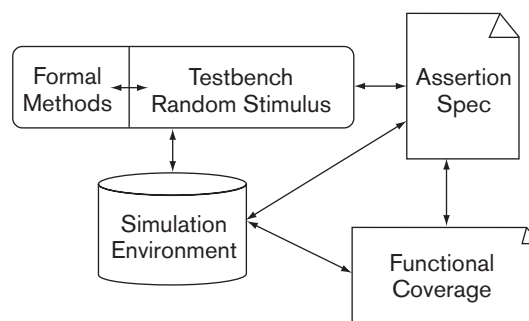


Figure 2. Reactive Testbench

Figure 2 illustrates a reactive test bench environment. Using functional coverage derived from assertion specifications as an input to the random stimulus generator, the algorithmic parameters are automatically modified to generate more effective tests.

2.4 Re-usable Assertions Save Time

The lack of assertion-based methodology of writing assertions, makes it impossible to re-use previously well tested checkers. That means the effort of writing a similar specification is duplicated from project to project, and from company to company. This has huge consequences are huge when standard protocols, such as PCI and FutureBus, are to be verified. It is clear that a capability to develop libraries of assertions can provide time-savings for many designers. Some examples are:

- Standard protocols
- Core designs that are the basis for derivative systems
- Commonly used assertions for design elements

Intellectual property (IP) vendors that ship pre-built designs can supply built-in assertions, thereby alleviating the quality concerns of customers. An assertion language verification IP has well defined and tested ready-to-use assertions for designers.

3.0 Synopsys' Assertion-Based Verification

Synopsys has developed a powerful new functional verification solution that includes a language, analysis tools and a methodology to support the concepts of assertion-based verification. This verification environment provides the following capabilities:

- Open Vera Assertion (OVA), a powerful new language to specify assertions
- OVA based bug detection, includes simulation as well as formal methods
- OVA based functional coverage
- OVA based test bench automation

This assertion-based solution is illustrated in Figure 3.

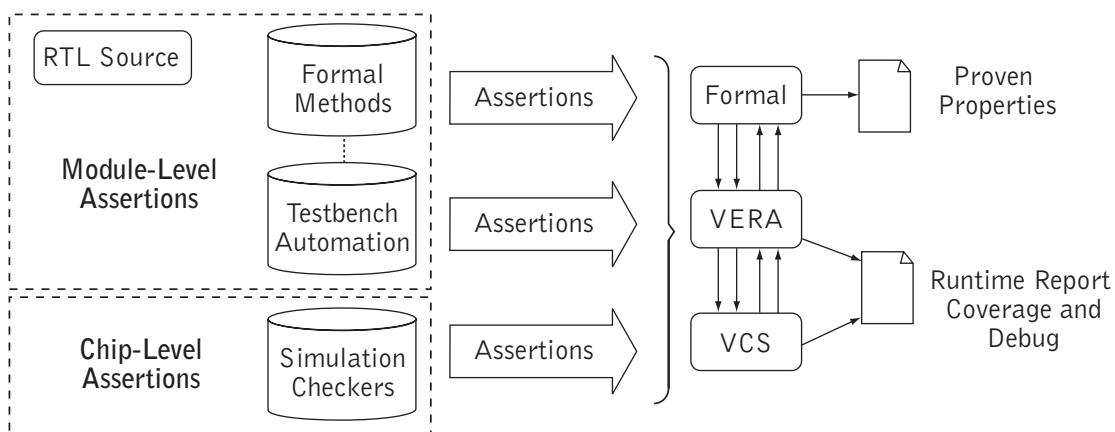


Figure 3. Synopsys' Assertion-based Verification

3.1 OVA Language

The OVA language has been developed to accurately and concisely describe behaviors that span over multiple cycles and modules of the DUT. For hardware developers, multi-cycle timing and order related functional problems are difficult to detect and the most expensive to fix, causing unpredictable delays and cost overruns. Commonly used hardware description languages (HDLs), such as Verilog and VHDL, were designed to model behavior at the register transfer level (RTL) (i.e., on single cycle transitions) and provide procedural features to describe the systems. Clearly, such an execution model is not sufficient to efficiently specify multi-cycle functions. Using OVA, users can easily and intuitively express input/output behavior, bus protocols, and other complex relationships of the design.

The OVA language is a declarative language whose semantics are formally based on the theories of regular expression and linear temporal logic. These two theories provide a powerful combination of expressing common hardware activities such as sequencing, invariants and finite state machine operations. Furthermore, the language can be translated to a finite state model that can be used by the formal methods to perform property checking.

Although OVA specifications express a wide range of behaviors, the following is a summary of language features that are applicable to most common situations.

- Basic events and sequencing to specify the order of events
- Time bounded sequences with references to past and future
- Complex sequences using logic connectives (such as and or)
- Repetition of sequences
- Conditional sequences with conditions to trigger or hold during sequences
- User specified single or multiple clocks
- Data storing and checking during sequences
- Parameterized library of specifications

The example in Figure 4 illustrates the expressive power of OVA.

```
clock clk {
    event seq_e:if (seq)then event1 #[min_time ..max_time ] event2;
}
assert timed_seq:check(seq_e);
```

Figure 4. A simple example of OVA

In our example, assertion `timed_seq` ensures that when the triggering event `trig` occurs, a sequence of `event1` followed by `event2` happens in a time window of minimum time `min_time` and maximum time `max_time`. Here, one statement describes a parallel behavior as the multi-cycle event `seq_e` is evaluated at every clock tick. Moreover, `seq_e` itself comprises of parallel paths related to the time range between `min_time` and `max_time`. Such a description in HDL would require several complex parallel blocks of code.

One of the most unique strengths of OVA is the ability to abstract behaviors arbitrarily. Such expression capability enables test benches to be configured and viewed at the appropriate level according to the test plan. For example, the bus protocol read and write events in Figure 5, are described at the lowest level using the protocol signals.

```

// define basic expressions
bool set_up_read: PSEL && !PENABLE && !PWRITE;
bool enable_read: PSEL && PENABLE && !PWRITE;
bool set_up_write: PSEL && !PENABLE && PWRITE;
bool enable_write: PSEL && PENABLE && PWRITE;
bool idle: !PSEL && !PENABLE;

clock posedge PCLK {
  event read: if (set_up_read) then #1 enable_read; // defines read operation
  event write: if (set_up_write) #1 enable_write; // defines write operation
}

assert a_read: check(read);
assert a_write: check(write);

```

Figure 5. An example of low-level description

Now, the events in Figure 5 are used to construct assertions abstracted at a higher level as shown in Figure 6. The events and assertions describe the behavior of idle cycles of the bus protocol.

```

clock posedge PCLK {
  event IdleCycle_1: read#1 idle #1 read;
  event IdleCycle_2: read#1 idle*[1..10] #1 read;
  event IdleCycle_3: read#1 idle #1 write;
  event IdleCycle_4: read#1(*mark = i4; data = addr*) idle*[1..10] #1 write;
}
assert a_IdleCycle_1: check(IdleCycle_1);
assert a_IdleCycle_2: check(IdleCycle_2);
assert a_IdleCycle_3: check(IdleCycle_3);
assert a_IdleCycle_4: check(IdleCycle_4);

```

Figure 6. An example of higher level abstraction

Another important feature of OVA is its ability to facilitate re-usable specifications. This is accomplished by parameterizing assertions and building a library of re-usable descriptions, such that the descriptions can be used multiple times in different DUT configurations, or in entirely different products. The savings in verification resources are not only substantial, but guarantee the quality of pre-built components. When the design models are supplemented by an assertion specification for design models, IP vendors who supply these design models would be crossing a big hurdle of current customer anxieties over quality concerns.

The example in Figure 7 illustrates the definition of a template `general_seq` that defines a re-usable description. Template `general_seq` is instantiated as `read_access` and bound to the specific design signals used for the instance.

```

template general_seq(clk,trig,event1,event2,trig_end,min_time,max_time):{
  clock clk {
    event seq_e: event1 ->> event2;
    event timed_e: length [min_time ..max_time ] in (seq_e #1 trig_end);
    event timed_seq_e: if (trig)then #1 timed_e;
  }
  assert timed_seq: check(timed_seq_e);
}
general_seq read_access(mclk,req,read,ack,req_t,1,4);

```

Figure 7. Example of assertion re-use

3.2 Bug Detection

OVA provide a very effectual mechanism to detect bugs. Quality, being the essence of functional verification, is improved at a much faster pace with greater likelihood that the design verification effort would meet the assurance on the quality of the finished product. The analysis software is built into the Verilog simulator VCS,[™] to provide instant reporting of violations when any assertion fed to the simulator is found to conflict with the expected behavior. Using the VCS simulation engine, the complex algorithms to monitor the assertions work in close conjunction with the simulation data and use the VCS insight into events as they occur to determine the assertion results.

Owing to the close cooperation with the VCS engine, the speed of simulation is minimally impacted by the assertion analysis, giving users feedback at a rate that they are already used to with VCS. This speed of analysis enables users to run large regressions that normally span over several hours to days.

OVA specifications may be written in a separate file to facilitate test plan implementers who are focused on the specifications at the chip level, rather than the design implementation details. To allow designers the capability of expressing assertions within the code itself, OVA can also be specified in Verilog as in-lined assertions. This freedom of in-lining assertions in the code speeds up design unit level testing. Once written in the source code, VCS ensures that the assertions are satisfied, no matter in which environment or application the code is used.

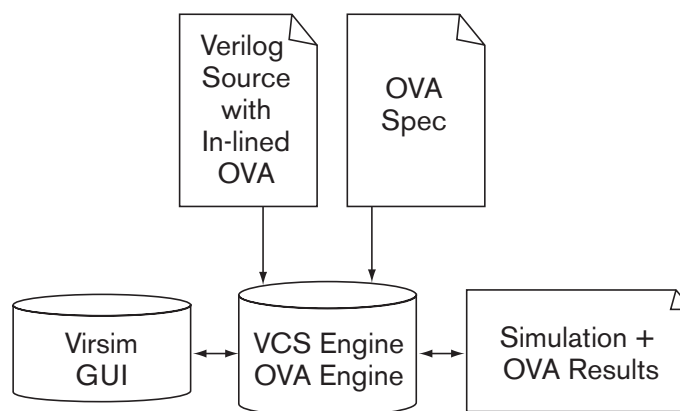


Figure 8. Ease of use with VCS

As shown in Figure 8, the built-in assertion analysis in VCS presents a user flow that is preserved without causing additional burden on the users to learn multitude of steps in getting the assertions to execute. The same approach of uniformity and ease of use is carried over to graphical debugging of problems and deeper analysis. Virsim, a graphical analyzer for VCS, directly shows assertion successes and failures at the same level as their corresponding assertion specification in a familiar waveform context.

The ability to detect failures is also provided by formal methods. OVA specifications do not enforce what applications can interpret the precise description of design behavior. Taking advantage of this independence, formal methods are applied to the same description. Even though the mainstream application of assertions is to verify designs using simulation, OVA language extends full support for formal verification as well.

A user can take an OVA specification that was used for formal verification over to simulation, and vice versa. This capability of moving between simulation and formal verification is extremely important for two reasons.

1. Given the state of art in formal verification, only a subset of assertions can be applied to formal verification, with the rest being left for simulation to validate. It is not always possible to know in advance through analysis which assertions can be verified by formal methods with a given set of resources. Considering this shortcoming, OVA allow a user to write assertions that are independent of the methods by which the design is verified.
2. More and more semi-formal techniques are being deployed that co-operate between formal and simulation methods to produce more efficient verification solutions. In such cases, consistency of results between formal and simulation methods are essential for semi-formal techniques to work, and require a common specification as provided by OVA.

3.3 Functional Coverage

There are three types of measures that are critically important for functional coverage:

- Test plan coverage, where the coverage indicates whether the expected behavior corresponding to each test in the test plan is covered.
- Implementation coverage, where the coverage indicates whether the checkers written to ensure correctness of design implementation are exercised.
- Data and statistics coverage, where the coverage ensures that in addition to control, data is adequately exercised and specific statistical counts are reached.

The OpenVera' Assertion language (OVA) is designed specifically to ease abstracting design functions by providing language constructs to build a hierarchy of concise temporal expressions representing events, sequences and transactions. Functional coverage is based on this hierarchy of expressions that enables the tools to measure coverage on higher levels of design functions. The level of abstraction itself is flexible and is determined by how a user chooses to compose and build a hierarchy of smaller sequences representing functions. Using OVA specifications, tools can clearly identify design functions at any level of abstraction that get covered by tests and others that need to be covered.

As in OVA support for formal verification, a user need not re-write complex expressions for functional coverage. Assertions that express design functionality, invariants and other behaviors are taken as the groundwork on which the coverage measurement is built. For insight into how each of these assertions is exercised, a user may augment the specification with coverage requirements.

For instance, it is not sufficient to know that a packet that arrived at the input was received at the output within the constraints set forth by the design specification. There may be several stages of transformation that the packet travels through before arriving at the output. Which combinations of those stages did the packet exercise is a more detailed knowledge that the functional coverage provides to guide the test plan implementation.

The example presented in Figure 9 uses the events specified in Figure 6, and supplements with additional coverage specification. The coverage data for the bus is collected during the time window implied by event IdleCycle_4. All executed paths are accumulated by the coverage function full_paths. A mark named idle4_end is introduced in event IdleCycle_4 to identify a point in time where address values are to be collected. Coverage function sample collects the addresses whenever the marked point is reached in the sequence.

```
event IdleCycle_4: read #1 idle*[1..10] #1 write (* idle4_end *);
coverage bus: IdleCycle_4{
  idle4_paths: full_paths(IdleCycle_4);
  idle4_addr: sample(addr, idle4_end);
}
```

Figure 9. An example of functional coverage specification

Functional coverage data is collected and presented at the level of abstraction at which the assertions are written, to minimize the manual work of correlating data between the signals' values and design behavior. Construction of data sets as abstract data objects operating on sequences, rather than the signal values depicted in terms of simulation time, is an example whereby examining the output of functional coverage is simplified. Data sets are sampled at any arbitrary point specified by the user within a sequence.

3.4 Re-usable Components

One of the great benefits of writing assertions in OVA is that the specification can be reused with no additional effort. OVA provide features to describe parameterized descriptions, which means you can not only customize assertions, but also the functional coverage specification. The parameterized description is instantiated to realize the goals, based on the environment or application in which the description is used. Examples of what you can customize are:

- Bind the actual design signals to the parameters
- Redefine clocking expression to suit the environment
- Redeclare sizes of variables
- Specialize coverage goals
- Filter coverage data sets to suit the environmental conditions

4.0 Productivity gains Using Assertion-Based Methodology

This section presents key points of how verification productivity is increased by employing a methodology centered on assertion specifications.

4.1 Creating a Test Plan with One Assertion Specification

A design functional specification, created as the first step in the design process, lays the groundwork for writing a test plan. The test plan, written in English, is a comprehensive plan to ensure that every identified design function is exercised and sufficient means exist in the test bench to selectively exercise a combination of design functions. Directed tests are then defined for each identified function, while random testing is planned for a combination of design functions. The test plan, is further enhanced, by inspecting the design implementation plan and devising tests to validate design behavior, that impacts the overall operations of the system. Thus, the test plan provides for testing externally visible functions, as well as any behavior that is known to be important from the implementation.

This test plan is an English language document that is manually translated to the OVA specification. This is obtained by abstracting the function from each directed test and writing an OVA event to check for the correct behavior and which coverage would adequately indicate that the design function targeted for the test is exercised. For example, a read operation on a bus may involve several signals, but the relationship between request, grant and end signals may be sufficient to indicate that the read operation was exercised. A series of OVA specifications would then correspond to the directed tests in the test plan. For random tests, a combination of functions can be expressed by relating individual OVA events expressing basic functions to compose more complex interactions.

The verification described here supports both “black box” and “white box” testing approaches. Both types of assertions are expressed in OVA. In black box testing, no knowledge of the implementation is assumed. The tests provide stimuli at the inputs and check the response at the outputs. Using the input/output pins of the DUT, the relationships between the inputs and outputs of the design are expressed as OVA events. The OVA specification remains independent of the internals of the design and is associated with the design without any modification even though the design itself may go through refinements and quality improvements. Verification engineers check for the correct system behavior, and write assertions (or checkers) to detect system-level bugs. In particular, checking the adherence of interface controllers to the interface protocol is important as these often hide hard to detect errors.

In OVA, the user can also access internal design signals to define events for the white box related tests. To verify a complex design, checkers are written at various levels of the design. Generally, designers having intimate knowledge of their design write assertions to ascertain that illegal behavior is trapped and notified as an error. These assertions are expected to hold true during the execution of the entire test plan. OVA can be conveniently specified in the source code where the assertion is expected to hold. Coverage of such events ensures that the corner cases, based on the detailed design and implementation knowledge, occur as a result of running those tests. In this way, module or unit level verification is accomplished.

4.2 Coupling Test Plan Assertions and Functional Coverage

Synopsys' assertion-based verification tightly couples assertions to check functionality and monitor functional coverage by using a common OVA specification. As a result, assertion checking and coverage run hand in hand to provide a clear view of the verification status. Figure 10 illustrates the process for creating a comprehensive test and coverage plan.

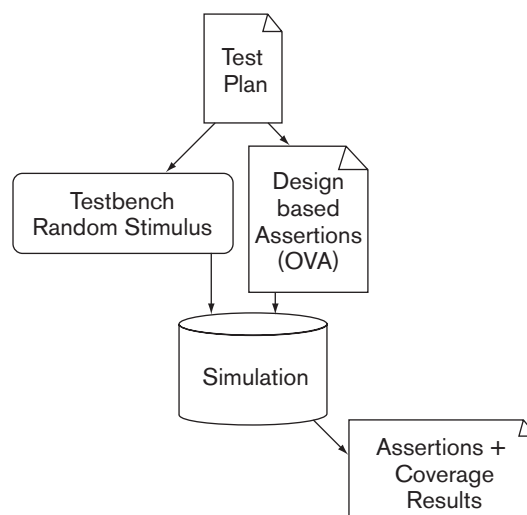


Figure 10. Functional Test and Coverage Plan

Once the system or a design module is simulated, the assertions must be exercised. The coverage capability monitors the success/failure of each assertion, because simply knowing whether an assertion was exercised is not sufficient. The assertion may succeed in multiple ways by taking certain paths (or branches) within the assertion. Each path covers a particular execution scenario. Functional coverage enumerates the paths and monitors them. At the end of simulation, the results give a scorecard against the enumerated paths to show how extensively the assertion was exercised.

The primary goal of system testing is to uncover bugs resulting from the interactions of various components in the system. During this process, there are numerous activities of interest to monitor that may not directly indicate a bug, but provide guidance into what must be examined to determine if there's a bug, inadequate testing or poor code. These activities are sequences of events or data values during a specific period. Often, statistics about data values and sequences are gathered as simple counts or frequency of occurrence.

OVA, supplemented with coverage constructs directives, provides the capability to detect a sequence or a sub-sequence within a sequence, and mark any point during a sequence for sampling data values. A sophisticated mechanism to collect this information for presentation allows extensive features for filtering and selecting of coverage data. A user can also utilize any OVA description already written for a test by appending a coverage specification to gather detailed coverage data.

The user obtains the data at the end of simulation, and merges the data gathered from the entire test suite. The data and statistics are compared against the goals set for the frequency and counts of data occurrences to determine if the tests need improvement, or the code needs a more thorough examination for possible bugs.

4.3 Test Bench Automation

Assertions written in OVA improve test bench automation for directed tests and random tests. A directed test is devised to address a specific feature, design functionality or design behavior. These individual test items addressed by directed tests are adequately verified by running a suite of directed tests.

However, many of the common design behaviors that interplay between multiple individual features are too numerous to be included in the suite of directed tests as coding them manually in the form of directed tests is not practical. Also, writing tests for selective interactions between features is not effective, because it is extremely difficult to predict which interactions are prone to failure. To account for testing behaviors resulting from a combination of features, random testing is employed, whereby a simulation run continues with random inputs, constrained only by the legalities of the input protocol of the DUT.

To make directed tests easy to implement, the test bench obtains feedback from the occurrence of events and sequences monitored by the OVA assertion engine. A test written in Verilog or a hardware verification language (HVL), such as Vera, simply initiates a design function and then waits for its subsequent completion indicated by an OVA event.

At the completion of a design function, the test may verify the resulting data through an algorithm or by comparing against a known response. This automation step is derived from the same assertion code that is written to validate a design behavior and obtain functional coverage. Using this technique, transaction based test benches are made simpler to implement because transaction control is already modeled by the associated assertions.

Random tests, whose objective is to exercise a combination of features and catch corner cases, become significantly more effective when the knowledge of which features or functional scenarios are getting exercised is given to the randomization algorithm. Taking advantage of the functional coverage information as the simulation progresses, the randomization algorithm modifies the constraints on the inputs and adjusts the probabilities of the input stimuli to improve the chance of exercising combinations that are different than previously exercised.

The formation of reactive testbenches using the functional coverage feedback is the goal of test bench developers who need to automate the testing process to cope with the exponential growth in the complexity of interactions between design functions.

OVA, with a systematic approach to assertions, simplifies fabrication of reactive testbenches, as the homogenous environment created by the assertions offers direct links to simulation, functional coverage and checkers.

5.0 Conclusion

Moving verification to the functional level is necessary as designs grow to multi-million gates. There are many aspects of functional verification: assertion specifications, test bench automation, coverage, simulation and formal property checking. Synopsys' Assertion-based verification achieves functional verification by building a common functional level source to drive these verification tasks.

Assertion-based verification is attractive because it takes advantage of various advances that have been made in verification, such as transaction-based test benches, temporal assertion checking, and functional coverage. As a methodology, it is a comprehensive start-to-finish approach, which not only integrates various tools, it executes them intelligently to accomplish a well specified common goal.

SYNOPSYS®

**700 East Middlefield Road, Mountain View, CA 94043 T 650 584 5000 www.synopsys.com
For product related training, call 1-800-793-3448 or visit the Web at www.synopsys.com/services**

Synopsys, and the Synopsys logo are registered trademarks and VCS is a trademark of Synopsys, Inc.
All other products or service names mentioned herein are trademarks of their respective holders and should be treated as such.

All rights reserved. Printed in the U.S.A.

©2003 Synopsys, Inc. 3/03.TM WO