

DesignCon 2006

Automated Risk Elimination By Formally Critiquing Verification Plan And Design Documentation

Jeff Li, Superior Logic Corporation
jeffli@inclusivesim.com

Abstract

There are always risks of missing bugs in functional verification. Using a formal analysis engine with very high capacity, we are able to eliminate such risks by formally proving that it is safe not to run a given class of test cases. This risk elimination solution works well with all verification flows because it only requires the waveform of a transaction and some user interaction (indicating what will be well verified about this transaction) in addition to the Verilog RTL. If it finds any test case in the class that can show any surprise, it generates a Verilog testbench to use in a normal simulation/debugging environment.

Author(s) Biography

Dr. Li is a verification solution architect. He also provides consulting services. His experience includes ASIC/SoC/IP/microprocessor design verification and software development in many programming/scripting languages. His degrees are in Computer Engineering and Electrical Engineering.

1. Introduction

Engineers can make mistakes in the design process of a VLSI device. Such mistakes are possible in writing a design document (which presents the design intent) and in coding in a hardware description language (HDL). Functional verification (with simulation or with the real hardware), following engineers' understanding of a verification plan (which presents the verification intent), is supposed to catch functional mistakes in the design document and in the HDL code. The quality of the verification plan depends on the engineers' understanding of the design document, and is generally determined only by visual inspection. As the verification technology becomes more and more reliable than visual inspection, the quality of both the verification plan and the design document becomes a more serious issue. In this paper, we pioneer the trend of developing automated technologies to improve the quality of these documents. We focus on engineers' understanding of the verification plan, which is closely related to engineers' understanding of the design document.

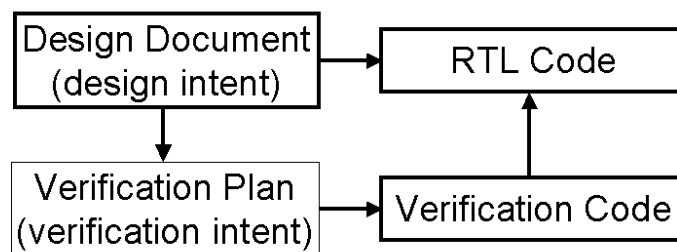


Figure 1. Verification plan is traditionally the weak link in assuring functional correctness because all other links have objective quality measurement methods.

Generally, a design document is supposed to be complete and flawless, but a verification plan is not meant to be perfect. Because of the resource constraints, any verification plan has to balance between the cost and the quality. Therefore, various elements in a verification plan's quality are sacrificed for either or both of two reasons: (1) the high cost and (2) the low impact on the product quality. Due to the routinely compromised quality of the verification plan, accidental flaws can stay undetected in the verification plan and in the design document, and these flaws can result in functional bugs in the VLSI device itself. In this paper, we only focus on where the cost is perceived as high and the impact on the product quality is also high, and we also address accidental flaws (in the documents and in the device) as a byproduct.

The correctness of an operation in the device can have relatively low impact on the product quality if the error can be automatically corrected. In many cases, the error correction capability is in the software that works with the device. An example is the ability to resend lost packets in some communication systems. Our solution here gets the definition of the interesting operations from users without any automatic estimation of the

impact on the overall product quality. When users define the interesting operations to verify, they should rely on the relevant descriptions from architects and designers. The engineering team should decide the impact level of each operation's correctness on the overall product quality.

A verification plan is usually rather good at covering parts of low verification cost even if the verification plan is simply an informal checklist. An important aspect of a part's verification cost is how easily people remember to include this part in the verification plan. Because a design document is usually good at including all intended operations, a cross-reference table should ensure the inclusion of all intended operations in the verification plan. Therefore, it is generally low cost to run a few basic cases of each intended operation.

Table 1. Classification of functional bugs to catch: prioritized coverage in verification plan

	Low Cost / Well Covered	High Cost / Neglected
High Impact	reference test cases (already in verification plan)	test cases under investigation (to add into verification plan?)
Low Impact	optional	optional

A design document normally describes clearly what's supposed to be involved in an intended operation in the device, but it can only imply what's not supposed to be involved in the operation. As the design document is supposed to be complete, everything is not supposed to be involved if it is not explicitly described as involved – unless there is a mistake in the document. It is traditionally hard to catch such mistakes in the design document because it has generally been high cost to verify what's not involved in an operation. A reason of the high cost is the difficulty to exhaustively identify all these implied parts of the design document. Another reason is the huge number of such implied parts in the design document for a modern VLSI device. Due to the huge size of a modern VLSI device, many operations can happen at the same time with different operations in different parts of the device. It is often important to also consider the impact of earlier operations to later operations. Although certain mistakes in such involvement can have high impact on the overall product quality, a verification plan often completely ignores these issues due to the inability to address them sufficiently. It is the focus of this paper to address such issues by providing a low cost solution.

There exist many solutions to reduce the verification cost, but they have not been proven sufficient to address these high cost verification tasks that are traditionally neglected in verification plans. Randomized simulation can find many problems in these high cost parts, but its verification completion criteria generally do not include many high cost verification tasks [2, 4, 5]. Hardware-assisted verification is similar. Assertion-based verification can help, but there is not yet any methodology for it to conclusively complete many of these high cost tasks. Formal verification improves verification efficiency dramatically where it works, but it is so far not generally applicable to many tasks related to chip-level verification plans [1]. There is no evidence to show generally that any

combination of these technologies can catch all important functional bugs with a reasonable amount of resources.

Our solution to address high cost verification issues is not a general verification solution because it depends on other solutions to cover the traditionally low cost verification tasks. It does not provide any meaningful results without these other solutions. Therefore, its most apparent application is to evaluate verification plans, where low cost verification tasks are covered sufficiently well already. By focusing on verification plans, it may possibly identify certain mistakes in design documents and in the device under test indirectly.

Our solution massively proves equivalence relationship between test cases, and the details of these test cases do not need to be specified separately. It can quickly show whether all test cases for a “high cost” verification task are equivalent to the test cases that are already in the plan. If any is proven otherwise, a Verilog testbench is generated to show this test case for analyzing it with any normal debugging environment. The analysis result may show it as a good case to add into the verification plan, it may show a mistake in the design document or in the user’s understanding of the design document, or it may show a functional bug in the device.

Our solution is low cost also because it is easy to use. It does not require any coding or drawing. It reads the waveforms (in a VCD file) and the RTL source, and it only asks users a few specific questions about how the verification plan is related to the information from these files. Therefore, it can be used with any mixture of verification methodologies.

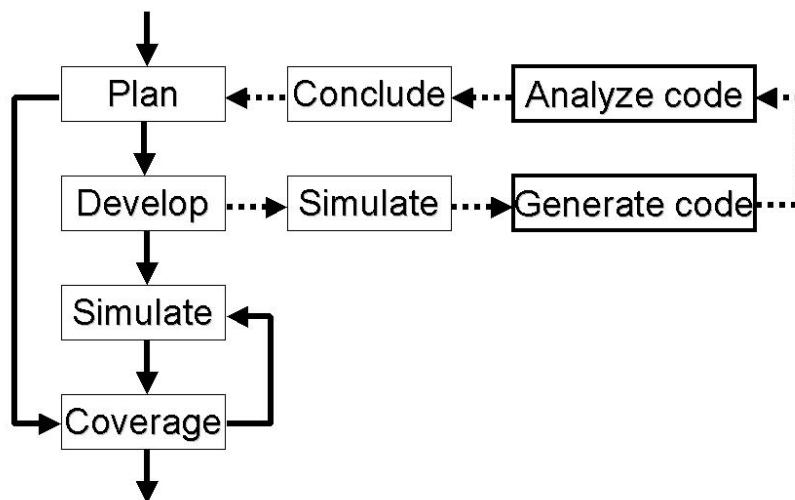


Figure 2. The new branch (dotted lines) can be easily added to any existing verification flow. The two “Simulate” steps can often be performed as one.

Due to its speed and ease of use, our solution may also be used to reduce the cost of some verification tasks that are already in or need to be added into the verification plan. It can

be applied to the “low impact” parts of the verification tasks if needed though it runs faster and is easier to use if focusing more narrowly on the “high impact” parts.

Our solution to a class of problems is presented in the next section. Its expansions in many directions are also discussed in this paper, including solutions that require some small amount of coding.

2. Basic Approach

Our solution includes 2 automatic steps, shown as generation and analysis in both Fig. 2 and Fig. 3. The first automatic step specifies the reference test cases, the test cases under investigation, and how to compare these 2 sets of test cases. The second automatic step statically analyzes the comparison model between the 2 groups of test cases. User actions are required at the beginning and at the end. There are also optional user actions in the middle. This section is organized mostly according to what users do.

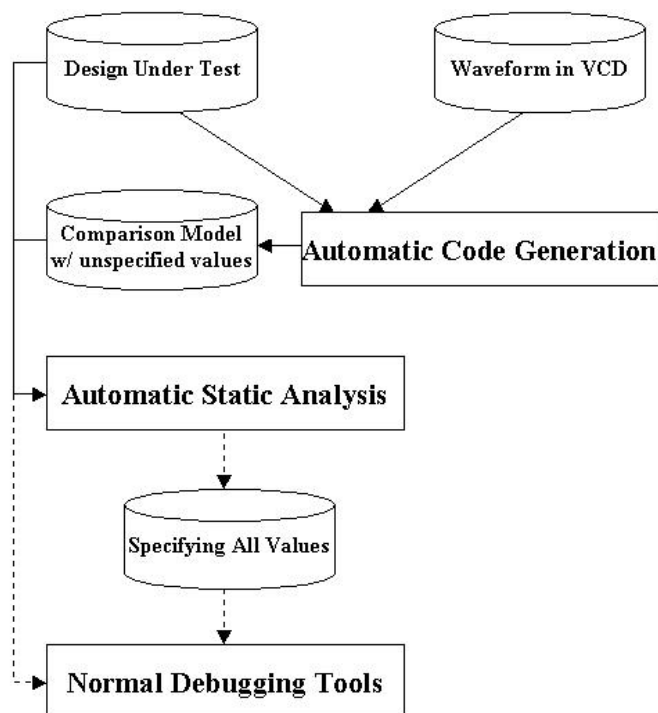


Figure 3. All involved files are Verilog or VCD. They are either automatically generated or always available. Manual changes in them are possible if needed.

2.1. Background

The ideal quality of a verification plan is to catch all important functional bugs, and the ideal quality of a design document is to define what are bugs and what are not. These are

their interesting aspects relevant to our discussion here though there are other aspects for other purposes. Due to the inability to sufficiently cover certain “high cost” areas, a verification plan in the past usually did not try to achieve this ideal quality. As the result, the design document’s quality was not fully validated and the design’s functional correctness was not fully verified either.

We are able to provide a low cost solution to cover the “high cost” areas because of the assumption that there is no need to cover many of the “high cost” areas in the conventional way. This assumption is consistent with the past experience in the industry because most devices turned out to be good though their functional verification never fully covered all “high cost” areas.

In widely accepted terms, this assumption indicates that applying a good set of corner cases is as good as applying all possible cases in functional verification. This assumption is widely accepted though the clear definition of corner cases is not as widely known. Brand [6] gave a good definition of corner cases for this purpose with a method to find all corner cases.

Given this assumption, our solution can be used to validate an existing verification plan, which fails to cover certain areas due to the high cost. Our solution may determine whether the existing test cases in the plan are a good set of corner cases that already cover these neglected “high cost” areas. If not, our solution will conclusively approve the plan only after adding a rather limited number of test cases. Here, covering an area means being able to catch all functional bugs of a certain class.

Now let’s look at a specific type of “high cost” areas first. According to a typical description of a basic operation in a design document, the values at a given source location at a given time is propagated (or transformed in a given way) to a given destination location (at a given time or at a time indicated by a given signal value) if the control values are set as the given pattern. This description does not include many signal values in the device, and it does not discuss many values of the discussed signals if these values are not for the discussed times. The “silent statements” imply that these “supposedly irrelevant” values should never keep the operations from completely following this description. Verifying the correctness of these “silent statements” is typically not well covered in a verification plan due to the high cost although some incorrect behavior in these parts can cause some serious failure in the device. Structural analysis is not good enough generally because frequently the control values conditionally keep these “supposedly irrelevant” values from disturbing the basic operation.

Our solution starts dealing with these “silent statements” by checking whether they are (directly or indirectly) covered by the test cases already in the verification plan. The test cases already in the verification plan are used as the reference test cases, and the test cases under investigation are the test cases that directly cover these “silent statements”. If these 2 sets of test cases are able to catch the same functional bugs, the verification plan is already good enough. Otherwise, the verification plan needs to be expanded. The

required information for comparing these test cases includes how to stimulate and how to observe the basic operation.

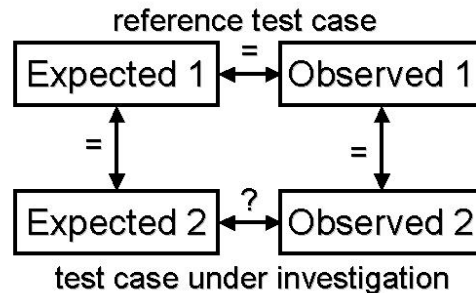


Figure 4. Indirect coverage: if two test cases show exactly the same behavior, the known success of the reference test case implies the success of the other.

Let's take a 2-to-4 demultiplexor (demux) as a trivial example. The 2 input bits are A and B. The 4 output bits are M0, M1, M2 and M3. The design document clearly states how the output bits depend on these 2 inputs bits, but it does not say anything about another input bit C. As the result, the verification plan includes all 4 input patterns (000, 001, 010, and 011) with C=0 while observing all 4 output bits, but it includes no input patterns with C=1. In this section, we will look at whether any input pattern with C=1 can catch any functional bug in this demultiplexor.

```

always @ (a or b or c) begin
    m0 = ~b & ~a;
    m1 = ~b & a;
    m2 = b & ~a;
    m3 = b & a;
end

always @ (a or b or c) begin
    m0 = ~c & ~b & ~a;
    m1 = ~c & ~b & a;
    m2 = ~c & b & ~a;
    m3 = ~c & b & a;
end
end

```

Figure 5. Two different implementations of a demultiplexor. Only one is correct or both are correct. The planned 4 test cases cannot catch the difference (bug?).

2.2. Stimulation & Observation

The verification intent (the information in the verification plan) is presented to the first automatic step as how to stimulate and observe the DUT in the test cases. A VCD file generated from a reference test case makes this presentation very simple.

Given the waveforms of a complete basic operation in the device under test (DUT), users can select a set of waveform segments to indicate that the permutations of different values for these segments are all included in the reference test cases (i.e. the test cases already in the verification plan). We will call this set S1.

Users must select another set of waveform segments to indicate that they are supposed to be the ones (and the only ones) determining (i.e. truly having impact on) this basic operation's behavior according to the design document. This set defines the apple-to-apple correspondence between the test cases under investigation and the reference test cases. We will call this set S2.

Each waveform segment has a signal name, a value and a time interval. These 2 sets can only include register values (including memory array contents) in the beginning of the basic operation's duration and primary inputs values for any parts of the duration. Each waveform segment's time interval is a cycle of the main system clock.

Frequently, S1 is a subset of S2, but they can also just partially overlap. Therefore, S1 sometimes has 2 parts: one inside S2 and the other outside S2. Normally, there is no need for the part outside S2.

There could be another set of waveform segments indicating what are included in the test cases under investigation. However, it is not needed because all waveform segments outside S2 should normally be considered in the test cases under investigation.

For observing the basic operation, users specify pairs of a time interval and a signal name. We call these pairs the observation selection. We can also call each of these pairs a waveform segment, but we do not do so merely because the actual signal value in the waveform segment is not relevant here. The observation selection (as well as S1 and S2) is used to present the verification plan, and the verification plan has no information about the observation results except the expectation. The signals can be any internal signals, including those connected directly to primary outputs. The times are all relative to an optional triggering event specified by the user as a given binary constant appearing on a given signal within a given timing window. If this event is not specified, the times are all relative to the beginning of the basic operation's duration.

The observation selection determines how the basic operation's behavior is considered correct. Normally it can include all waveform segments that are dependent on S2. If only certain parts of the behavior are considered critical, the observation selection should include at least these critical parts. For example, the data correctness may not be as critical as the ability to handle the next operations if the system can always automatically recover from data corruption.

Our solution automatically parses the VCD file and the top-level RTL file so that it can provide all required information for users to select all the above in a graphical user

interface. This VCD file has to contain initial values of all internal registers and primary inputs' values for the basic operation's duration.

None of the required information is from the testbench because the verification plan and the testbench are supposed to match. This separation from the testbench makes our solution neutral to verification methodology choices. It also allows applying our solution before completing the testbench.

In the demultiplexor example, the waveform segments for A, B and C are made available to select from the VCD of simulating one of the 4 patterns. To indicate all 4 patterns in the verification plan as the reference test cases, the waveform segments for A and B are selected as S1. To indicate the only supposedly relevant inputs, the waveform segments for A and B are also selected as S2. The waveform segment for C (i.e. C=0 in the original VCD) is not in either set because C=1 is not used in the verification plan at all and C is not considered relevant to this demultiplexor. There is no need to select any initial register values because the demultiplexor is supposed to involve no sequential elements at all. All output bits (with the time just after applying the input pattern) are selected for observation to indicate the relevance of all 4 bits.

2.3 Generated Comparison Model

Given the details of the stimulation and of the observation, our solution automatically generates a Verilog file that represents the problem to be analyzed. It is also fine to manually create this file or manually edit it if more flexibility is needed.

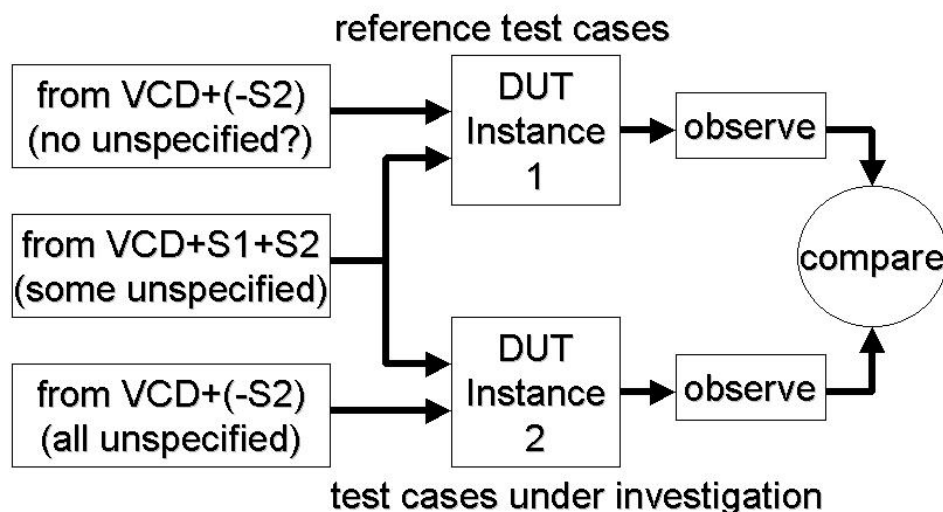


Figure 6. The Verilog model for comparing test cases. It can be either manually written or automatically generated based on the VCD file and user selections.

This Verilog file basically contains a Verilog module. This Verilog module includes 2 identical instances of the DUT. It also hooks the 2 instances to the stimulation logic and

to the observation logic. The 2 instances of DUT are observed independently in the same way, but they are stimulated partially together and partially in different ways. All waveform segments in S2 are applied to both instances in the same way, assuring that the differences between the 2 instances are supposed to be irrelevant to the basic operation being observed. Therefore, the correct comparison is enforced as the reference test cases stimulate the 1st DUT instance and the test cases under investigation stimulate the 2nd.

The stimulation side includes some constant values, and it may also include unspecified values to be specified in the next analysis step. All constant values are from the original VCD file. These unspecified values are determined by S1 or S2. The waveform segments outside S2 (if not given as another set of waveform segments) are all shown as unspecified values for the 2nd DUT instance. The waveform segments in S1 are shown as all unspecified values for the 1st DUT instance. Any waveform segments in both S1 and S2 (excluding those in one set but not in the other) are shown as unspecified values for both DUT instances. These unspecified values allow the next analysis step to cover many cases together, which is a key enabler of the dramatic cost reduction in our solution.

In the demultiplexor example, A and B have unspecified values for the 1st DUT instance because they are in S1, and all 3 input bits have unspecified values for the 2nd DUT instance because A and B are in S1 and S2 and C is not in S2. Only C has a constant value 0 for the 1st DUT instance because this is C's value in the VCD file.

```

// for instance 1
  initial begin // inputs
    `VPEstart;
    `VPEclkneg;// cycle 1 : #0
    a_1 = `VPEwildcards[0];
    b_1 = `VPEwildcards[1];
    c_1 = 1'b0;
  end
// for instance 2
  initial begin // inputs
    `VPEstart;
    `VPEclkneg;// cycle 1 : #0
    a_2 = `VPEwildcards[0];
    b_2 = `VPEwildcards[1];
  end
end

```

Figure 7. Stimulate the demultiplexor. Two unspecified bits are shared between the two instances. An input to instance 2 is not connected at all due to not in S2.

The constants and the unspecified values are applied to both DUT instances cycle by cycle. They are applied to internal registers (only the initial cycle) with assign/deassign procedural statement pairs, but not to any other internal signals (except memory arrays). They are applied to primary inputs of the 2 DUT instances possibly with switching among the constants and the unspecified values between the cycles.

In most cases, each primary input of each DUT instance has to be connected to a reg for using different procedural assignments for values in different clock cycles. Its unspecified values have to be assigned to this reg every cycle in an initial block if the unspecified values need to be different between the cycles. Therefore, the unspecified values all have to be represented as net bits, implying that they do not carry their values from one clock cycle to the next. For convenience, our solution puts all unspecified values into a single net vector if they are not in a net already.

For each waveform segment that is in both S1 and S2, the identified primary input (or internal register) of both DUT instances gets the same unspecified value. This is achieved by assigning the same unspecified value (as a part of the net vector) within the same clock cycle for both DUT instances.

For the following 3 cases, a primary input of a DUT instance does not need to be connected to a reg:

- If it always gets unspecified values according to either S1 or S2 (possibly both), it can be connected to a net (or not connected at all if none of its waveform segments is in S2).
- If it always gets the same constant value, it can be connected to this constant value directly or through a net.
- If all waveform segments for a DUT's primary input are in S2, this input can be connected to the same source (reg, net, or constant value, depending on S1) for both DUT instances.

In the demultiplexor example, C for the 1st DUT instance can be simply tied to constant 0. C for the 2nd DUT instance can be unconnected, or it can be connected to a net without any source. A for both DUT instances must be connected to the same signal bit, and B for both DUT instances must share another signal bit because A and B are in both S1 and S2.

The observation side uses a bit to indicate the success of the comparison. This bit can be a net or a reg. For coding convenience, the generated code records all observed data, according to the observation selection in the 2 DUT instances. The comparison is not successful until all needed data are stored and unless all recorded data match as expected between the 2 DUT instances. The 2 DUT instances are observed independently, both according to the same observation selection for assuring the required similarity between the 2 DUT instances' responses. To tolerate timing differences, the trigger to start observation is generated for each DUT also independently as needed. The window of the trigger's possible time is defined with an optional waiting loop and another waiting loop marking the 2 ends of the window.

In the demultiplexor example, the 4 output bits of each DUT instance are compared against the corresponding output bits of the other DUT instance. The comparison is successful only if the 2 DUT instances have identical output patterns. If the comparison is successful, all choices of the unspecified values on the stimulation side will cause the 2 DUT instances to generate matching output values, especially when C=0 for the 1st DUT instance and C=1 for the 2nd (given the ties of A and B between the 2 DUT instances).

2.4 Analysis Result

Because of the unspecified values, the generated comparison model cannot be used correctly with a normal Verilog simulator. The normal default values for a reg and for a net are x and z, respectively. The generated comparison model is supposed to be used like a blank form. It can be used with a normal Verilog simulator only after assigning binary constant values to all the unspecified values. The next automatic step is to select these binary constant values.

As the generated comparison model compares test cases against each other, automatic static analysis is used to determine whether the comparison is always successful. If it is always successful, there is no need to use the generated comparison model any further because the reference test cases (i.e. the test cases already in the verification plan) can cover all bugs that the test cases under investigation can.

If the comparison is not always successful, binary constants are assigned (cycle by cycle) to the unspecified values in the earlier generated comparison model. These new assignments are generated in a new Verilog module to make the earlier generated comparison model suitable for reading into any normal Verilog simulator. These assignments show an intriguing test case that generates unexpected behavior. Then the debugging work can be done with any traditional debugging tools, as shown with dotted lines (indicating only for unsuccessful comparison) in Fig. 3.

It is interesting to carefully examine how the DUT behaves when exercising this intriguing test case. It probably shows a bug in the DUT as the supposedly irrelevant logic is actually relevant. It probably shows a mistake in the design document (or in the user's understanding of the design document) because the relevant logic is not expected to be relevant. It may also be just a mistake that the user made when interacting with our solution. It is possible to show many of such intriguing test cases, but it might not be interesting to carefully investigate all of them because they might show the same problem.

In the demultiplexor example, binary constant 1 is connected to C for the 2nd DUT instance if C is proven relevant. Any 2 binary constants can be connected to A and B for both DUT instances so that the only different input value between the 2 DUT instances is C because C gets 0 for the 1st DUT instance as described in subsection 2.3. Therefore, by simulating the 2 DUT instances with C's value as the only difference, people can see how C is truly involved in the demultiplexor's operation. This is possible with the 2nd implementation in Fig. 5, and C is apparently not involved at all in the 1st implementation. For example, it can be misunderstood whether the need of an enable/disable bit is implied.

2.5. Performance and Experience

We use a special static analysis engine (not a formal or static verification tool) to perform the automated static analysis step. Many formal verification tools can be used for static

analysis, but they are not known to be able to handle the large properties used in our comparison models.

Among the successes, we are able to approve the verification plan of a full duplex Ethernet design though it does not include any test case with traffic in both directions. All possible activities (valid/invalid traffic/states) at the incoming interface are shown unable to disturb the normal outgoing traffic.

The speed of the automated static analysis step is often proportional to how intuitive the comparison is rather than how big the design is or how many test cases are involved. This is one of the reasons why we use two DUT instances for a big set of test cases under investigation. We usually have many thousand bits in the unspecified values, sometimes even close to a million bits. It normally takes minutes or less to finish, often resulting in speedup as high as many thousand orders of magnitude. It takes hours or more only if it can only use a very small percentage of the CPU time due to the insufficient physical memory. The worst case of the memory requirement is close to being linearly proportional to the number of the main system clock cycles in the duration of the basic operation. Therefore, it is a good idea to use short basic operations.

The automatic code generation step takes seconds, especially if the basic operation is short. It is unlikely for this step to take more than a few minutes even for very big designs.

Therefore, the most time-consuming parts are the manual steps, especially considering the big number of different basic operation types. Due to no requirement of either coding or drawing, these manual steps can still consume much less time than expanding the verification code. Due to the convenience of testing short basic operations, the debugging time is much shorter than debugging traditional long tests though the debugging tools are exactly the same. Therefore, our solution is better in all 3 major time-consuming parts of functional verification projects.

It is useful to automate more parts of the manual steps though it can be hard to do so without more specific knowledge about the project/device details. The graphical user interface is written in Tcl/Tk so that it can be easily customized according to specific project's needs.

3. Some Application Tips

The selection of S1 and S2 can in many cases sufficiently present what are in the verification plan and what are to be investigated. Because this may not be the most natural style to represent a verification plan, methods to handle various special needs in applications of our solution are presented in this section. With these methods and the extended application in subsection 3.5, it is possible to use some slightly modified versions of our solution to achieve all needed improvements in verification plans. Some of the methods involve more manual work than the above, but they are all easier than writing/customizing normal testbenches.

3.1. The Basic Operation and Catching Bugs

A test case in a verification plan may involve many operations. Some operations are parts of other operations. It is not always obvious which operation should be used as the basic operation in section 2.

Our recommendation is to focus on what bugs to catch. Look at all relevant test cases in the verification plan together. Among the many operations in a test case, some are the focus of the test case and others are just the required supporting functions. Hints can be found from what are observed in a test case. It is also helpful to understand what are verified in the previous test cases.

For example, many test cases involve setting control registers or loading data buffers. Are they able to detect failures of the setting or loading? Are these failures supposed to be all detected in some earlier test cases? It is often a good idea to apply our solution to a basic operation like setting control registers or loading data buffers. In such an application, it is important to specify what variations of the basic operation are used in all relevant test cases where the failure of the basic operation can be detected. Such an application is not needed if it is not important to learn whether anything else can cause a failure in setting the control registers or loading the data buffers.

Many test cases involve many rounds of using data buffer contents for certain purposes. Different rounds may use the contents of the same data buffer location (at different times) or of different ones. Is each of these rounds able to catch some failure that cannot be caught by any others of such rounds in the verification plan? Are any of the rounds only for bridging from one interesting operation to another? In most cases, the basic operation should be a single round. Shorter test cases are always known to save debugging time if they uncover bugs, and they are also known to improve runtime efficiency with our solution. Depending on the answers to these and other questions (some are in the later parts of this section), it may also be reasonable to use a sequence of the rounds as the basic operation.

3.2. The Basic Operation and Input Hand-Shaking

Instead of (or in addition to) using data buffers, many devices get data from primary inputs. Many of these cases are equivalent to connecting these primary inputs to some external data buffers. It does not cause much complication if these imaginary data buffers work consistently every time. However, it is different if these data buffers' interfaces can respond sometimes faster and sometimes slower.

Apparently, the selection of S1 and S2 does not support any way to indicate the timing variation. There are at least 3 ways to handle this situation:

- (1) The basic operation can be defined as starting with receiving the response on the primary inputs. In this way, the timing variation is at the end of the basic operation (or at the end of another basic operation), and therefore it does not interfere with stimulating the basic operation. The observation side supports

timing variation more conveniently as various timing windows can be easily added into the observation side of the generated comparison model.

- (2) Divide each test case with timing flexibility into multiple test cases without timing flexibility. This is not too much trouble when the timing flexibility is very limited.

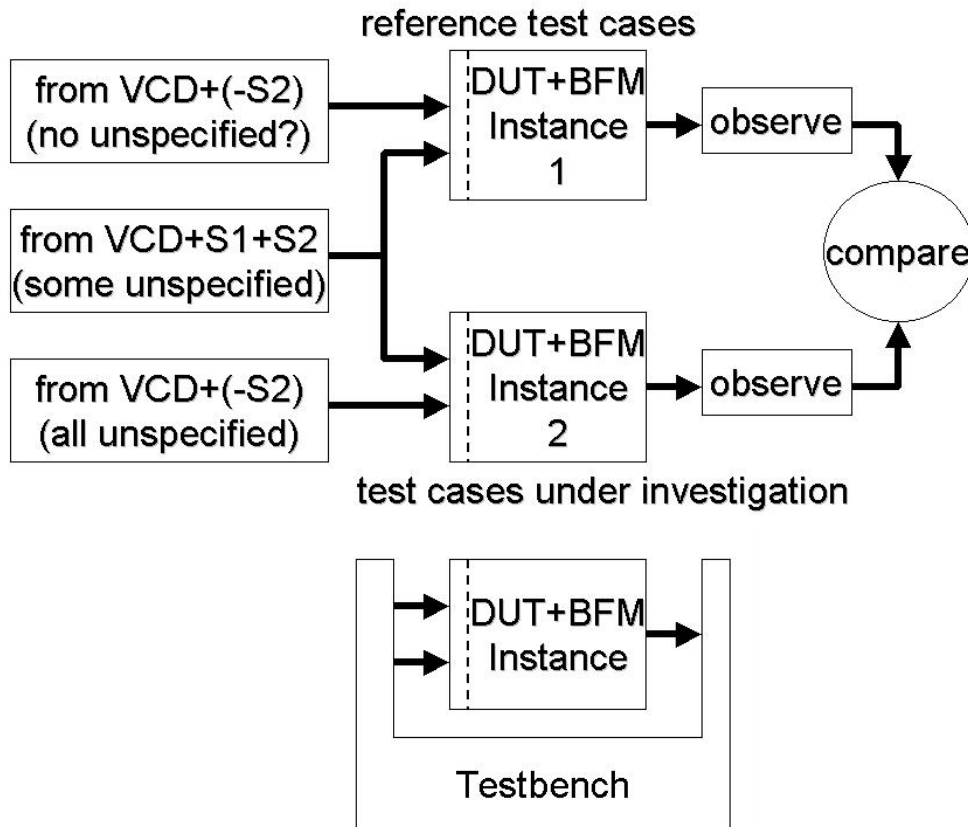


Figure 8. Top: Treat bus function models as attached to DUT. Bottom: Treat bus functional models also as attached to DUT in a normal verification environment.

- (3) There is normally some code feeding these primary inputs to handle the handshaking of this interface. Such code can be a part of a component called transactor or bus function model (BFM). It is possible to expand the DUT in the generated comparison model to include such code. Then the interface of the expanded DUT can be as clean as needed. In the transactor/BFM, some bits of unspecified values can be used to control the delay amount through something like a counter. This does not take much extra work because the transactor/BFM code is needed for normal verification as well, and it can be used for both purposes as shown in Fig. 8.

3.3. The Basic Operation and Validity of Initial States

The selection of S1 and S2 treats the initial values of registers in the DUT as similar to the primary inputs of the DUT. This is easily acceptable for data buffers, but it can be tricky for state variables. Because many of the state variables are not described in the design document, it is often hard to decide whether certain states are valid initial states for the basic operation.

Even if a state is claimed to be impossible, it is often still interesting to check whether it is really possible. It is interesting to do so especially if this state can lead to real problems. The details of some real problems are disclosed in [3]. On the other hand, there is no need to do such checking on a state if starting the basic operation with this state does not cause any trouble.

In our solution, most state variables can safely take unspecified values, and they probably do not even need to share the same unspecified values between the 2 DUT instances. If any of them makes a difference, the intriguing test case will show everything clearly, which may lead to uncover real bugs or to find more details of the design intent. This capability makes it practical to use our solution with short test cases, which are known easier to debug generally and faster to run with our solution.

When there is any real need, our solution can also be used to analyze whether certain states can possibly be reached within a given time window. This analysis may reach a conditional conclusion, and the condition specifies the sequence of operations and the states to start the sequence with. The analysis is completed if any of these new initial states is known valid. Otherwise, there is a need for another round of the analysis to determine the validity of the new initial states.

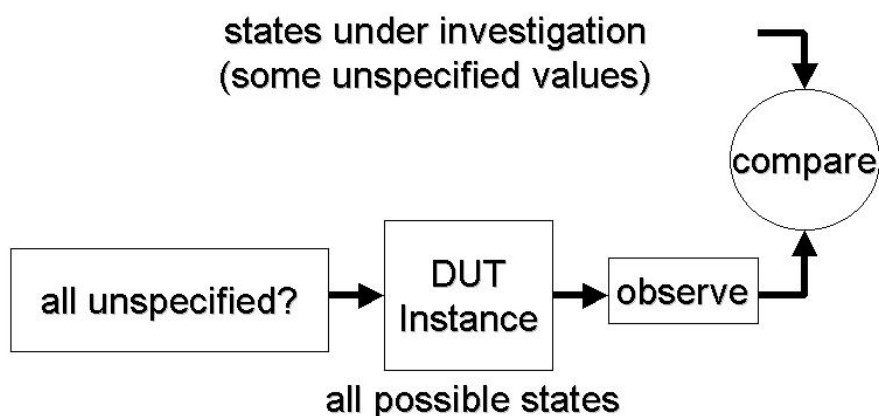


Figure 9. Analyze whether any of the states under investigation can be one of the possible states. This is to validate initial states when they cause real trouble.

When analyzing the validity of some given states, a DUT instance in the generated comparison model plays the main role and the other DUT instance plays the supporting role. The supporting instance should generate these given states, or it can be ignored if these given states are coded into the comparison logic. The comparison logic is coded to fail if the main DUT instance generates one of these given states. The unspecified values for the main instance should allow all possible operations, even including troublesome operations in many cases. Then the static analysis step of our solution may show an initial state and a sequence of operations to reach one of the given states. If it finishes with nothing to show, the given states (states under investigation in Fig. 9) are all invalid.

It is sometimes possible to avoid the initial states' validity issue by choosing a long sequence as the basic operation. This is the traditional style on most verification projects. If a long sequence and a short sequence are both reasonable candidates of the basic operation, the short sequence is normally the more efficient choice. The long sequence usually includes more bridging operations that consume resources and are equivalent to setting internal register values directly. The set of long sequences may be a bad choice also because of the difficulties with hitting some of the critical initial states.

3.4. Irregular Boundaries

The selection of S2 (or S1) creates a multidimensional cube in the binary space where each point is a test case. Frequently, smaller cubes (with less dimensions, including a point, a line or a square) need to be added to or excluded from this big cube due to some special conditions of the basic operation.

The simplest way to handle such irregular shapes is to use multiple cubes of various dimensions. Any irregular shape can be decomposed into points, and many of these points can be merged into lines, squares and cubes of various dimensions. Sometimes it takes too many cubes to cover the irregular shape, and sometimes it takes only a few. If it takes too many cubes, it may not be the best choice to split the basic operation into so many in this way.

As an alternative, sometimes it is a good idea to create a function that maps from a multidimensional cube to the irregular shape. This function may map 2 or more points in the cube to the same point in the irregular shape if needed. This function can be implemented in the assignments of the unspecified values in the generated comparison model (by changing the right-hand sides from simple identifiers to expressions). There are at least two equally powerful styles for any irregular shape: (1) identifiers in expressions are also used alone in the assignments and (2) identifiers in expressions are not used alone in any of the assignments. Of course, the shared stimulation between the 2 DUT instances must be maintained where it is needed.

As another alternative, it is possible to code the constraint into the comparison logic in the generated comparison model. The constraint has to specify exactly the points to be excluded from the multidimensional cube, and it has to assure the success of the comparison for all these points even if the 2 DUT instances behave differently.

The last 2 methods can handle S1 and S2 together, and so can the first method with careful consideration. This is important when excluding test cases from being under investigation. If this exclusion is implemented by shrinking S1, the excluded reference test cases go away with all of the corresponding test cases under investigation. If this exclusion is implemented by enlarging S2, some test cases under investigation are excluded for each and every reference test case together. It is important to consider whether these test cases under investigation are all to be excluded.

3.5. Enriching Reference Test Cases

The use of S1 and S2 to select test cases for comparison assumes exhaustively verifying the explicitly described part of the basic operation. This assumption can be considered true if the verification plan requires reasonably good verification of this part. All previous parts of this paper depend on this assumption, but here we discuss how to provide this assumption with our solution.

When it is needed to better verify the explicitly described part of the basic operation, our solution can help as well. For this purpose, S1 should be empty so that the VCD file provides a single reference test case. S2 should include everything (primary input value or internal register initial value) except the intended unspecified values. Therefore, the generated comparison model will compare all these test cases against the single reference test case, which must be simulated successfully already because of the generated VCD file.

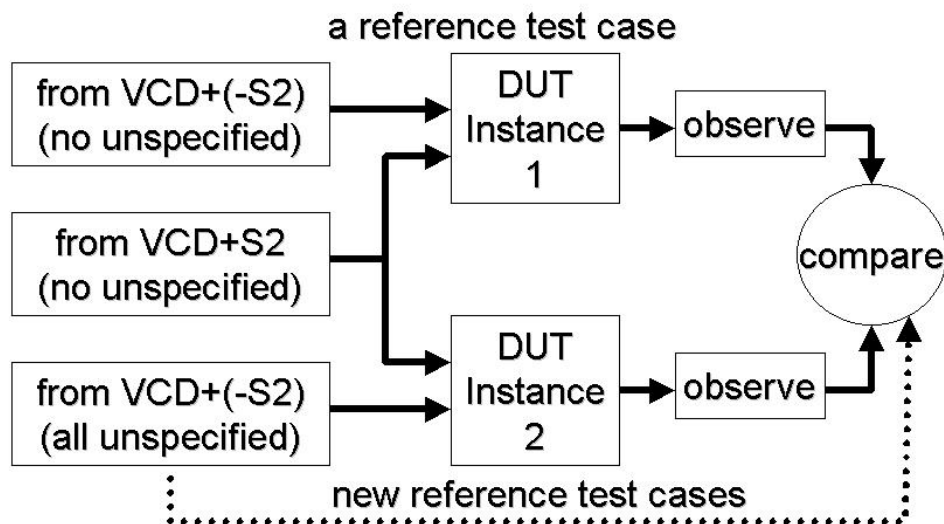


Figure 10. Verify the correctness of the DUT's response to the new reference test cases while the correctness of the response to a reference test case is known.

The tricky part in this case is the observation and comparison because the reference test case should show different behavior than the test cases under investigation do. The difference should be either considered or ignored in the comparison logic.

If the difference is to be ignored, the comparison logic needs to be coded carefully to include only the supposedly consistent part, e.g. the control/status outputs. This is sometimes enough to catch all critical bugs. In this case, even verifying the explicitly described part of the basic operation does not require any coding or drawing. The weaker comparison requirement turns most of the reference test cases into test cases under investigation, and therefore the benefits for test cases under investigation are also applied to these reference test cases.

If the difference is to be considered, the expected dependency on the unspecified values in the stimulation logic must be coded correctly into the comparison logic. Because the code at most needs only to describe the difference between the test cases, it is still simpler in general than traditional verification code, where the expectation needs to be generated completely. The runtime speed can also be much faster than simulation's but it can be slower than the basic approach in section 2.

Based on what is discussed so far, our solution should be able to cover all test cases except one reference test case per type of basic operations. A verification plan covering one reference test case per basic operation can now be proven able to catch all important functional bugs. This complete verification plan and the proof of its completeness may only require a reasonable amount of resources because of (1) the high speed and the reduced manual work in our solution and (2) the low debugging load for testing basic operations separately.

3.6. When to use it:

Our solution of critiquing verification plans and design documents requires the RTL code of the DUT. Therefore, it can be used only after the RTL code is available although it should be applied as earlier as possible. At least, its application should start before the testbench is fully implemented.

Due to its dependence on the RTL code, its result may change after any changes in the RTL code. Therefore, the initial round of its application is not sufficient. Another round is usually needed after the RTL code becomes really stable. It is also interesting to apply it after any significant changes in the RTL code or in the design document. Clearly, any significant change in the verification plan also introduces the need to apply our solution again if the reference test cases are changed.

4. Conclusions

It is possible for a simple verification plan to cover all important functional bugs. This can be true even if the plan only includes verifying each type of basic operations with a single test case. We have developed a practical solution using a special static analysis engine (not a formal or static verification tool) to conclusively determine such

completeness of a verification plan. If the verification plan is not complete, our solution suggests new test cases to make it more complete. Such a solution enables the truly successful completion of a functional verification project without risks, which has been impossible before partially due to the “casual” methods to critique verification plans.

Our solution is based on waveform data, 2 DUT instances and unspecified values. Its resource consumption is very low in coding, runtime and debugging, and it can easily help any existing flow.

Our solution’s application can be limited to the high cost (and high impact) part of verification so that it does not need to disturb the main verification flow (directed, randomized, formal, semi-formal, assertion-based, etc.). It depends on the main flow, but it has hardly any requirement on the main flow because it only needs the reference test cases’ waveforms from the main flow. It is often possible to reduce its cost even more significantly by limiting its focus to the high impact part of verification.

The required waveforms include all relevant input values and all relevant memory elements’ initial values. These values represent everything that determines the output values of the basic operation. Engineers need to identify all these relevant values, and also to identify which of these values are expected to change in the main flow for verifying different variations of this basic operation. They also have to identify the relevant output values, which can be internal signal values and primary outputs’ values.

With these values identified, there is no need for any coding or drawing, and a Verilog testbench can be automatically generated (with minor manual work in some special situations) with 2 instances of the DUT. Both instances share the same relevant values, but they do not share the supposedly irrelevant values at all. One instance is for the reference test case and the other is for the test cases under investigation. The generated testbench captures the relevant output values from both instances, and compares them.

In the generated testbench, unspecified values are used for some relevant values and, with 1 of the 2 instances, for all the supposedly irrelevant values. The special static analysis engine is able to process the testbench with the unspecified values quickly. It normally takes only minutes even if there are many thousand bits in these unspecified values.

If the analysis engine detects the ability for any of the test cases to catch different functional bugs, it generates assignments of binary constants to all unspecified values so that normal debugging tools can correctly process the Verilog testbench. These test cases of single basic operations are always very easy to debug due to the clear focus.

5. References

1. R.M.Gott, J.R.Baumgartner, P.Roessler, and S.I.Joe, “Functional formal verification on designs of pSeries microprocessors and communication subsystems” IBM Journal of Research and Development, vol. 49, pp. 565-580, September 2005.

2. A.Adir, H.Azatchi, E.Bin, O.Peled, and K.Shoikhet, "A generic micro-architectural test plan approach for microprocessor verification" in Proceedings of the 42nd Design Automation Conference, pp. 769-774, June 2005.
3. B.Wile, J.C.Goss, and W.Roesner, Comprehensive Functional Verification: The Complete Industry Cycle, 2005.
4. P.James, Verification Plans: The Five-Day Verification Strategy for Modern Hardware Verification Languages, 2004.
5. K.Albin, "Nuts and bolts of core and SoC verification" in Proceedings of the 38th Design Automation Conference, pp. 249-252, June 2001.
6. D.Brand, "Exhaustive simulation need not require an exponential number of tests" IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, no. 11, pp. 1635-1641, November 1993.