

# concepts

## Testbench Design, a Systematic Approach

### 1 Introduction

There are many technical publications in the electronic industry that have recognized and brought awareness to the key issue of functional verification that faces companies designing System on a Chip (SoC). This task is estimated, depending on the complexity and size of the chip, to take more than 50- percent of the total effort to bring the SoC to market. Table 1 below shows the main tasks related to the functional verification of a typical SoC.

Typically the verification engineer has to first study and understand the specification of the Design Under Test (DUT) in detail before any tests or testbenches can be developed. This paper suggests that if functional tests can be written in terms of the activities described in the specification of the DUT, these tests will be much easier to write, debug, and maintain. This can reduce the effort of developing and debugging tests by 25-percent (see Table 1), which currently at 60-percent is the largest effort required in SoC functional verification. Using the suggested architecture, the engineer can end up with a powerful what-if scenario engine that can lead to an increased level of confidence in the functional correctness of the SoC.

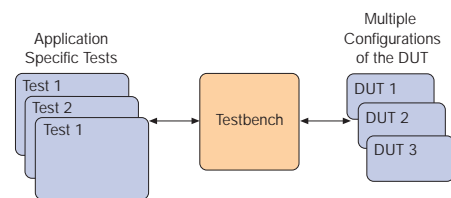


Figure 1. Generic Relationship between Tests, Testbench and DUT.

ID	Task Description for Functional Verification Only	Typical Effort Distribution for Main Tasks in SoC Verification *	Improved Effort Distribution Main Tasks in SoC Verification**
1	Study DUT specification and develop verification strategy	10 %	10 %
2	Develop detailed test plan	10 %	10 %
3	Develop testbench environment	10 %	35 % ***
4	Develop and debug tests	60 %	10 %
5	Run regression test suite	10 %	10 %
	Total	100%	75%
6	Effort savings	0	25 %

Table 1

\* Typical here implies functional verification using directed tests with non-layered approach and none to minimal use of self checking and randomization.

\*\* Improved here implies using the proposed testbench architecture methodology to achieve same functional coverage as stated in test plan.

\*\*\* This effort will be further reduced after the first project that uses this testbench methodology.

### 2 Testbench Architecture

Before discussing testbench architecture solutions, let's review the main goals of devising an improved testbench architecture:

1. Reduce the effort of test development.
2. Reduce the effort of debugging to isolate bugs easily
3. Enable re-use of tests from sub-system level testing in system-level testing
4. Provide quality metrics of the test through code and functional coverage
5. Maximize re-use of the testbench across multiple projects

To resolve this open-ended problem, it helps to begin by thinking at higher level. Tests are connected to the DUTs by the testbench, which in turn is connected to the DUT configurations as shown in Figure 1. Note that different applications require different tests and DUTs. Even within the same application, the same tests typically have multiple DUT representations.

The testbench needs a common architecture that takes into account the variations of the DUT as well as the variations in the required tests.

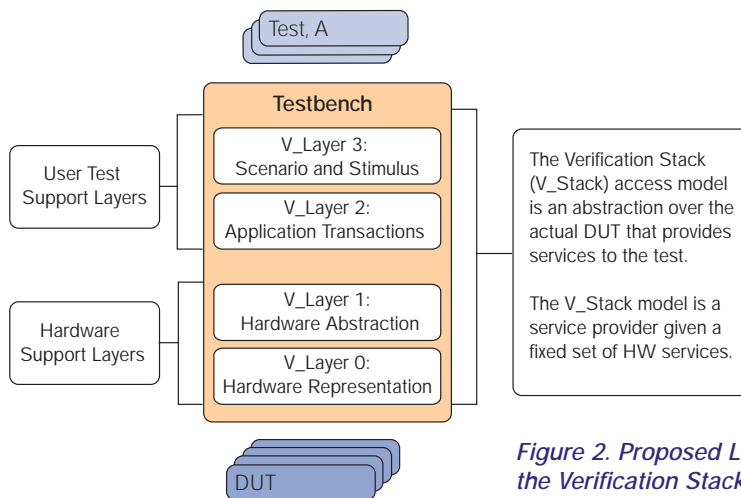


Figure 2. Proposed Layered Solution: the Verification Stack (V\_Stack model), with its four layers.

# concepts

## 2.1 A Layered Approach

The proposed solution suggests a layered approach with clearly defined functionality for each of the layers as shown in Figure 2. Each layer then provides a certain set of services to the upper layer, while shielding it from lower-level details. Each of these layers will be referred to as a Verification Layer (V\_Layer), and collectively, the four V\_Layers are known as the Verification Stack (V\_Stack).

### 2.1.1 V\_Layer 0: Hardware Representation Layer

This V\_Layer 0 provides signal-level connectivity into the physical representation of the DUT (typically described in HDL or SystemC™). For example, if the DUT is implemented in HDL and the rest of the testbench is in VERA®, this layer would consist of the VERA interface file. In it one would define the signals (of the DUT) that are of interest. In short, this layer provides signal name-abstraction and connectivity to the event-driven worlds of most simulation engines. Here is an example of the data in an interface file that constitutes this layer:

Signal Direction	Virtual Signal Name	Signal Sampling Specification	Corresponding Signal Name in HDL
Input	Core_clk	CLOCK	HDL_NODE "TB.dut.clk";
Output [31:0]	Xm_addr	Output_edge Output_skew	HDL_NODE "TB.dut.addr";

The concept here is that the test developer uses virtual signal names instead of real (implementation specific) names when implementing tests. This has the immediate and significant benefit that if the signal names or paths change in the DUT (or HDL part of the testbench), the tests would not be affected. It is not unusual for the interface layer to define several hundred signals, and for the (existing) environments to contain hundreds of tests. From a maintenance point of view alone, the verification engineer would have one less complication to deal with.

### 2.1.2 V\_Layer 1: Hardware Abstraction Layer

V\_Layer 1 provides a HW bus-abstraction view of the hardware. Its role is to make sure that the bus transactions issued by upper V\_Layers will reach the DUT, regardless of how the DUT is represented. Furthermore, this V\_Layer is responsible for configuring the DUT. Typically, this V\_Layer would include bus transactors, bus monitors, protocol checkers, and simple behavioral models for the case when the DUT is partially represented because RTL is not yet available.

Typical commands that are present in this layer usually correlate to bus transactions. Here is an example of a HW transaction that can be issued from this layer (PCI write, then read):

**Pci\_transaction (WE\_CMMD, ADRS, DATA, BYTE\_ENABLE);**

**WE\_CMMD:** a PCI bus command for write, and read.  
**ADRS:** an address in PCI address space.  
**DATA, and BYTE\_ENABLE:** data and the "byte\_enable" flags

### 2.1.3 V\_Layer 2: Application Transactions

V\_layer 2 provides the abstraction needed to carry out the operations of the SoC from the application point of view. This is the first layer that can reflect the actual services required by the application. Again, these services are defined from the application point of view and not from the test point of view. They also include the checking of results. With results checking built into the application transactions, an abstraction is provided that enables the test developer to focus solely on creating

the right stimulus, which can be either explicit or pseudo random based. This assures that once a transaction completes, the SoC also has processed it correctly.

For example, for the Security Processor chip, the SoC under test provides well-defined encryption services. Data encryption is handled by packaging the data inside a descriptor (similar to a packet). Descriptors can be dynamically dispatched to any one of the 9 channels, or statically dispatched to a dedicated channel. Therefore, from the application point of view, the application transactions consist of:

#### SET 1:

- Create & send a dynamic descriptor (Checking implied)
- Create & send a static descriptor (Checking implied)

When one does not include the checking in the application transaction, one gets a set of functions on a lower level of abstraction, for example:

#### SET 2:

- Create dynamic descriptor
- Create static descriptor
- Send descriptor
- Check dynamic descriptor
- Check static descriptor

This set of functions results in a situation where all of the complexities of checking the behavior of the SoC are moved up into the test, and therefore does not provide a clear self-containing abstraction. Having the checking embedded in the application transaction offers a clear interface that enables the test developer to focus on generating stimulus to the DUT.

# concepts

Unlike the bus-based transactions of V\_Layer 1, the transactions in V\_Layer 2 do not have a 1-to-1 correspondence with a HW unit. Instead, the application transaction is an abstraction that is carried out by the whole DUT and often the whole testbench. In the encryption SoC example, the create & send & check descriptor single transaction would generate a series of 16 PCI write operations that would be issued by accessing bus transactions from V\_Layer 1.

### 2.1.4 V\_Layer 3: Scenario and Stimulus

This V\_Layer provides high-level interface to configure the DUT, testbench, and the test. It also enables stimulus generation (manual or automatic streams). Streams are defined in terms of a series of application transactions (V\_layer 2 transactions). The test writer has control over various streams of application transactions by applying constraints to each stream, and/or using inter-stream synchronization based on system events.

Inter-stream synchronization is essential for defining complex test scenarios where the engineer needs to synchronize the flow of a stream with one or more activities in the DUT, or in another stimulus stream. This synchronization is done by System Events where the engineer can use well-known system events in the design. For example, an event can be defined by the PCI monitor to announce detection of bus abort.

This layer supports built-in self-checking at the inter-stream level. This stream-level self-checking augments the self-checking that takes place in each of the application transactions defined in V\_Layer 2.

### 2.2 The Testbench Architecture

With the above description of the intended functionality of the V\_Layers in mind, Figure 3 shows a general architecture of the V\_Stack. The figure

shows the objects that are typically present in each of the layers. The User Test is part of the object called Scenario Manager. Note how the User Test only interacts with the entire testbench

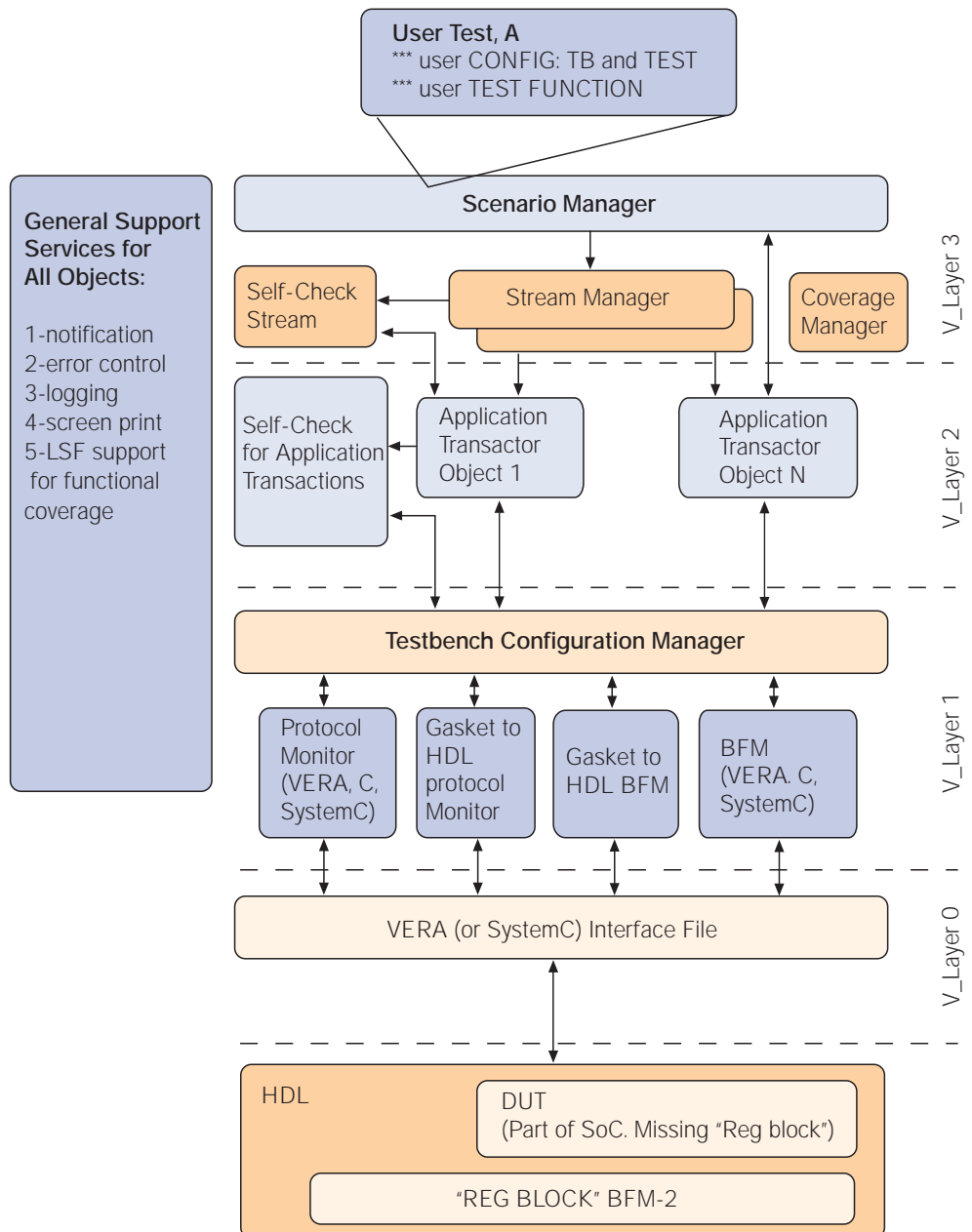


Figure 3. A Proposed Testbench Design that Takes Advantage of the Layered Approach.

# concepts

environment by the Scenario Manager. This object is the main user interface to this testbench environment. The User Test usually consists of two short sections: the configuration section (a set of knobs to adjust) and the test function.

At simulation runtime, based on the user configuration, the Scenario Manager instantiates all the objects necessary to carry out the test. Inside the test function, the primary activities are

- Issue application transactions
- Control streams of application transaction
- Request notification to any system vent

This architecture has been successfully used on two different projects, and other projects have begun to deploy it. Readers interested in a more in-depth treatment of the concepts presented and the proposed testbench architecture can refer to the web for a [white paper on this architecture](#).

## 3 Conclusion

The proposed testbench architecture is not an out-of-the box solution for SoC verification. It is however a systematic blueprint that substantially reduces the design and implementation time of the testbench. (Our experiences find more than a 25-percent reduction). Specifically, the architecture includes:

- The Scenario Manager (V\_Layer 3) that reduces the effort of creating tests.
- The TCM (in V\_layer 1) that allows re-use of sub-system tests at system level testing.
- The distributed self-checking techniques (V\_Layer 1, 2, and 3) to make it easier to isolate bugs.
- The functional coverage manager that provides the mechanism to track functional behavior against detailed test plan.
- The support services (with object-oriented programming techniques) which maximize the benefit of re-use across multiple projects.

This layered testbench architecture can be adapted to fit most digital designs. To gain the most benefit in effort savings and maximize re-use, the use of a powerful HVL like VERA with object-oriented capabilities and powerful concurrent thread (and corresponding synchronization) capabilities is highly recommended.