A C/C++-based Functional Verification Framework using the SystemC Verification Library

Sanggyu Park and Soo-Ik Chae

Center for SoC Design Technology,
School of Electrical Engineering and Computer Science
Seoul National University, SEOUL, KOREA

{sanggyu, chae}@sdgroup.snu.ac.kr

Abstract

This paper describes SoCBase-VL, which is a C/C++ based integrated framework for SoC functional verification. It has a layered architecture which provides easier testbench description, automatic verification of bus interfaces and seamless testbench migration. This framework does not require verification engineers to learn other verification languages as long as they have sufficient knowledge on both C/C++ and SystemC. We have confirmed its usefulness by applying it to a TFT-LCD Controller verification.

1 Introduction

In the dynamic verification, a set of stimuli is applied to a design and then, its responses are compared to the corresponding correct outputs to check its equivalence or correctness. This verification approach requires a testbench that generates stimuli and checks correct outputs. Thus, the quality of verification depends on the quality of the testbench. As the designs are getting more complex, however, the difficulty of authoring the testbenches is continuously growing even more rapidly.

The difficulties related to the testbench design can be summarized as follows:

- As the number of the state in a component increases linearly, the number of test cases increases exponentially. Therefore, manual enumeration of each test case is not feasible.
- Several models for a component may be required at different abstraction levels. A testbench for each model should be re-designed to verify the model.
- Describing a testbench often requires a deep and thorough understanding on domain-specific knowledge. e.g. Bus Specification.
- A quantitative measure of the quality of verification is needed. Otherwise, the quality of verification tends to depend on that of verification engineers.

To alleviate those problems, many researchers and EDA vendors offer tools for testbench authoring [1-5]. The SystemC Verification Library (SCV) is an extension of SystemC for easier testbench authoring which provides constrained randomization and transaction level tracing[1].

SoCBase-VL is another extension of the SCV, which additionally provides a layered architecture for easier testbench description, seamless testbench migration, and an automatic verification of bus interfaces. It also provides the Coverage Monitor Modeling Library (CML) for functional coverage monitoring.

In this paper, we explain our layered testbench architecture in Section 2 and the CML in Section 3. In Section 4, we briefly introduce how to use our framework through a practical example. The summary and future works are given in Section 5.

2 C/C++-based Layered Testbench

A H/W component (or a system) can have several abstraction level models: transaction level model, RT-level model, FPGA prototype and Silicon. We propose a layered testbench architecture, depicted in Fig. 1, by which a single testbench description can be used for verification of all models without manual modification.

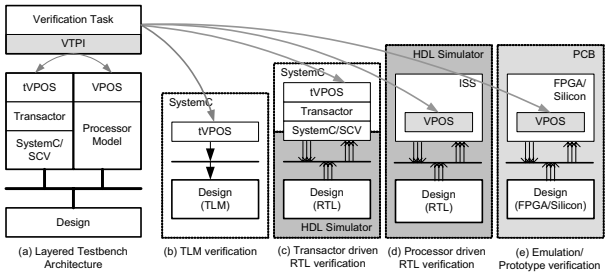

Fig 1. Layered testbench architecture

A verification task (v-task) is a software program that generates stimuli to a design and validates its responses. It is a C/C++ program described in Verification Task Programming Interface (VTPI), which is a set of functions and

macros for v-task description. A v-task is a software program and some execution methods are needed to execute it. There are two kinds of execution methods: the processor model and the transactor. The conceptual roles of both methods are identical, but their implementations are different. To abstract out details of the execution methods, we developed a verification-purpose light OS kernel (VPOS) and its transaction level version, tVPOS. The VPOS provides just the basic OS functions such as multi-tasking, memory management and interrupt handling. A v-task can be run on a processor model with the help of the VPOS or on the transactors with the help of the tVPOS, as shown in Fig 1(a).

A transactor is an abstraction adapter that translates a transaction function call into the bus signal activities, and vice versa. In our framework, we added into the transactor an important feature for the automation of the bus-level verification. The augmented transactor has two internal layers: a protocol layer and a signal layer, as shown in Fig. 2.

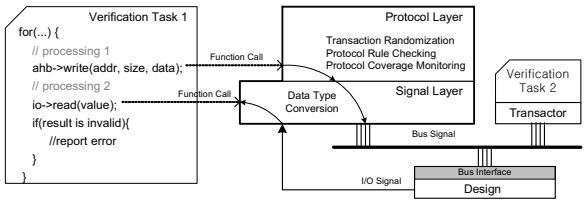

Fig 2. Internal layers of the augmented transactor

In the protocol layer, read (or write) function calls are translated into a sequence of transfers. Although a v-task can explicitly select the transfer type of each transfer, it also allows the transactor to select each transfer types randomly. In contrast, for the processor-driven testbench, bus-related parameters are ignored. Therefore, the v-task, which invokes the transactor, can control and accelerate the verification of the bus interface. This layer also checks protocol rules and provides the bus protocol coverage information. Because the protocol layer of the augmented transactor abstracts out the bus-level details, the v-task can be easily described without any bus-level details. Moreover, v-tasks of a component can be reused for verifying the same component with a different bus protocol.

The signal layer of the transactor translates each transfer to the bus signal activities and converts an abstract data types to a signal-level data type. For example, the integer type is converted into the std_logic_vector type in this layer. In the augmented transactor depicted in Fig. 2, the I/O signals in the RT-level design can be relayed to and from the v-task only though the signal layer. This feature enables the v-task to validate I/O signals which are not bus related.

Because a v-task can control the random behavior of its transactors, the verification engineer can set them to generate only simple transfers at the early verification stage, and then, change it to generate more complex transfers on later stages. Through this approach, we can easily localize the bugs related to the bus interface.

3 Coverage Monitor Modeling Library

We developed the Coverage monitor Modeling Library (CML), which is a C/C++ class library for functional coverage monitor description. Fig. 3 is a part of an AHB slave coverage monitor described using the CML.

```
cov_value NONSEQ, SEQ,IDLE, BUSY;                                    (1)
cov_value BYTE, HALF, WORD;                                          (2)
cov_literal HTRANS = IDLE|BUSY|NONSEQ|SEQ;                           (3)
cov_literal HSIZE = BYTE | HALF | WORD;                              (4)
cov_expr expr1 = (HSEL[0] == TRUE) & (HTRANS == NONSEQ)              (5)
        & (HREADY[0]==TRUE) & (HWRITE[0] == HWRITE.all()) &          (6)
      (HRESP[1] == HRESP_OKAY);
cov_expr expr2 = HTRANS[0] == (NONSEQ | SEQ)                         (7)
cov_expr expr3 = HTRANS[0] == (NONSEQ + SEQ)                         (8)
cov_sampler NOWAIT;                                                  (9)
if(HREADY == TRUE) { NOWAIT = HTRANS; ... }                          (10)
expr4 <<= NOWAIT;                                                    (11)
```

Fig 3. CML-based Coverage Monitor

In CML, five object types are defined: the coverage value, the coverage literal, the coverage expression, the sampler, and the coverage monitor. A coverage value represents a value of a state in the design and a coverage literal represents a state in the design. A set of coverage values must be assigned to a coverage literal. As in the expression (3), HTRANS can have any value among IDLE, BUSY, NONSEQ and SEQ. A coverage expression is a formal representation that defines multiple functional covers with coverage literals, coverage values and coverage operators which are listed in Table 1.

Table 1. Coverage operator

Left	Op.	Right	Meaning
V_A	|	V_B	V_A or V_B
V_A	+	V_B	V_A or V_B, respectably
L[k]	==	V_A	If the value of L[k] is V_A then hit
L[k]	!=	V_A	If the value of L[k] isn't V_A then hit
L[k]	>,< >=,<=	V_A	If the value of L[k] is {greater, smaller, equal or greater, equal or smaller } than V_A then coverage hit
E_A	&=	E_B	If the E_A and E_B is true then hit
E_A	+=	E_B	Merge the E_B into E_A

V_A : Value A L[k] : Literal[k] E_A : Expression A

The notation of literal[k] represents *the value at time k*. For example, the expression (6) reports a coverage hit when the current value of HREADY is 'FALSE' and the next-time value is 'TRUE'. Note that the function of '+' operator is somewhat special in defining the functional

covers. Expression (7) defines one functional cover which reports a hit when the value of HTRANS is NONSEQ or SEQ. Meanwhile, expression (8) defines two functional covers: one for a hit when the value of HTRANS is NONSEQ and the other for a hit when the value of HTRANS is SEQ.

To analyze each coverage expression, a set of history of the literals needs to be stored. A sampler is an object that stores the literal values cycle by cycle. Expression (9-10) defines a sampler that stores the literal values only when the current value of HREADY is TRUE. A coverage expressions must be linked to a sampler with an operator '<<=' as shown in (11). A coverage monitor is a collection of these objects that provides several common methods for user interface.

4 TFT LCD Controller Verification

In this section, we present a TFT LCD Controller verification example to show the effectiveness of our framework.

The basic operation flow of the TFT LCD Controller is as follows: 1) The AHB slave interface receives mode setup commands and configures the operation mode. 2) The AHB master interface reads in image data and stores those into FIFO. 3) The timing controller retrieves pixel data from FIFO and drives TFT LCD controller outputs. The TFT LCD controller has six parameters such as color format selections, bit inversion mode, and endianness. Therefore, it has a total of 64 operation modes to be verified.

The testbench architecture for the TFT LCD controller verification is illustrated in Fig. 4. The AHB slave interface of the TFT LCD Controller is connected to the v-task with a AHB slave transactor. And the AHB master interface is connected to the verification memory with a AHB master transactor. The TFT LCD controller output is connected to a SystemC panel model that receives the output stream of the TFT LCD controller and stores it to the verification memory. The v-task contains a C behavior model of the TFT LCD controller, and its six parameters are randomly selected and configured. The v-task compares the output of the RT-level model stored in the verification memory with the results of behavior model to validate the behavior.

From this test environment, we could verify the RT-level model of the TFT LCD controller thoroughly. Especially, the bus interfaces are verified without any manual description. The verification quality was reported with the AHB coverage monitors described in the CML. These monitors are reusable for other verification works. The v-task can be compiled with the compiler for the embedded processors such as ARM processors, and it can be run on those processor models with the VPOS.

With this feature, we could verify the TFT-LCD controller integrated in a FPGA prototype that includes an embedded processor by adding a hardware circuit that stores the output of the TFT-LCD controller into the memory.

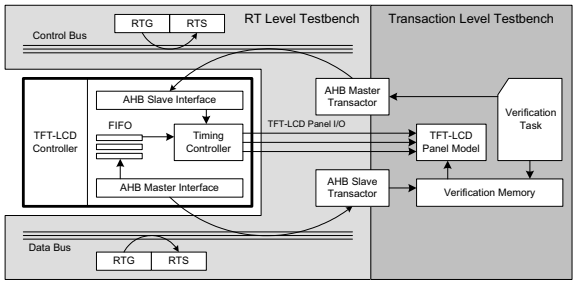

Fig 4. TFT-LCD Controller Testbench

5 Summary and Future Works

In this paper, we proposed an integrated framework with a layered architecture of the testbenches, which provides seamless testbench migration with a verification purpose operating system, the capability of precise and concise functional coverage monitor description, bus-level verification automation, and high-level testbench description power. Although each technique in the proposed framework is widely used, it provides a unified C/C++ and SystemC based framework. Our work on the verification framework is not finished yet and still on-going. Although the current version of the framework does not support software verification issues, we have a plan to enhance the VPOS for HdS verification. We also need to support the transactor-driven testbench in the emulation level by developing a more flexible emulation system in the future..

References

[1] Verisity, Inc. "Invisible Specman Developer's Guide Version 4.0.5", http://www.verisity.com

[2] Synopsys, Inc. "OpenVera Language Reference Manual", http://www.open-vera.com, Apr, 2003.

[3] OSCI, "SystemC Verification Standard Specification Version 1.0e", http://www.systemc.org, May, 2003.

[4] Cadence. "TestBuilder User Guide", Aug, 2003.

[5] ARM LTD, "AMBA Compliance Testbench User Guides", http://www.arm.com, Feb, 2003.