Overview of Object-Oriented Approach to HDL-Testbench Construction for System-on-Chips

Vladimir Hahanov, Dmitriy Melnik, Oleg Zaharchenko, Sergey Zaychenko

Abstract - **the goal of this paper is to provide a basic overview of efficient and powerful approach to testbench construction using an abstract object-oriented framework. Basic testbench environment is reviewed; several corner cases are demonstrated, like synchronization of several transactors working in parallel under SystemVerilog simulation toolsuite**.

Keywords – **Testbench, Object-Oriented Design, System-on-Chip.**

I. TRADITIONAL VS OBJECT-ORIENTED TESTBENCH ARCHITECTURE

Traditional directed testbench consists of several typical framework steps: generate transaction stimulus – apply stimulus to the DUT – obtain DUT response – compare outputs with expected values, check the correctness – measure the progress of the verification goals. Directed tests are mostly indented to check a single item of the verification plan, which is prepared basing on requirements specification. Reaching 100% specification coverage with tests means preparation of individual test for each function.

With traditional testbench framework the focus on relevant things like a testbench structure and component responsibilities is lost between large number of individual pin-to-pin stimulus and testbench-DUT synchronization patterns. Simple testbench is hard to debug, maintain and extend, it's re-use is limited, it has to be almost completely rewritten in case of interface changes. Coupled with continuous design complexity growth, verification complexity using directed testbenches grows exponentially. While being quite new in relation to HDL-testbenches, the Object-Oriented Design (OOD) approach serves for many years in software engineering and testing areas [22]. Like in software, the key principles of dealing with both hardware design and verification complexity with OOD are *partitioning* and *encapsulation*. Building abstract interfaces, minimizing dependencies between verification goals, hiding details in smaller blocks – all these steps lead to faster achievement of 100% functional coverage, improved visibility of verification progress, reduced test maintenance cost, facilitation of test components re-use and finally better product quality.

OOD allows to create self-contained complex data structures with related routines that manipulate with them.. In such higher-level testbenches routines are called to perform an action, rather than toggling bits [1]. Themselves those encapsulated routines directly communicate with interface signals and manage synchronization between DUT and testing environment. However it becomes much easier to maintain complex DUT activations, as elementary operations are being implemented and verified only once, and than can be re-used multiple times in various testing scenarios. An encapsulated activation procedure, performing driving of particular individual bits with values at certain moments of time accordingly to protocol, is referred as *transaction*. Dealing with transitions brings new level of productivity – testbench is separated from the design details and it is much easier to maintain and reuse it.

Objects and modules are quite similar things – it brings up an opportunity to use classes within a hardware verification context. Verification components can be successfully represented with instances of classes – it allows to great flexibility of classes paradigm in connection with real hardware components. Following table reflects comparison of basic OOD paradigms in traditional C++ classes and Verilog-SystemVerilog:

TABLE 2.2.1.
COMPARISON OF MODULES AND TRADITIONAL CLASSES

Feature	**C++ class**	**Verilog module**	**SystemVerilog module**
Inheritance	yes	no	Yes
parametrization with types	yes	no	Yes
dynamic objects	yes	no	Yes

The SystemVerilog virtual interface is one of the basic constructs that makes possible simple connection of object-oriented testbench framework to the hardware design. A virtual interface is a reference to an interface. A class can be written containing references to items inside an interface that doesn't yet exist. When the class is instantiated, the virtual interface is connected to an actual interface instance. This makes it possible for a class object to both drive and respond to pin activity [1].

Interfaces have two «sides» - one side of the interface provides functionality and the other one requires the functionality. Such «requires-provides» paradigm is the basis for connecting verification components at the transaction level [12]. Following is a «hardware-oriented» illustration.

```
module require( input CLK, input DATA_INIT, output DATA_PROC );
  ...
endmodule
```

The «require» module needs some signals and when theys are received at the right time and in the right order (then the module responds). The pin interface for the «provide» module supplies all the things needed to bind to the «require» module:

```
module provide( output CLK, output DATA_INT, input DATA_PROC );
  ...
endmodule
```

The behavior level modeling raises the abstraction of the block functionality, whereas transaction level modeling raises the abstraction level of communication between blocks and subsystems, by hiding the details of both control and data flow across interfaces [3].

II. CORNER CASES IN OBJECT-ORIENTED TESTBENCH APPROACH

Transaction level modeling is a concept, and not a feature of a specific language [2]. But there are some language constructs that are very useful for writing transaction-level models:

– structural hierarchy;
– function and task calls across hierarchy boundaries;
– the ability to package data with subprogram calls;
– the ability to parallelize/serialize data;
– semaphores to control shared resources;

A fundamental capability is to be able to encapsulate the lower level details of the data and control exchange protocols into function and task calls across an interface [2]. This complies an object-oriented interaction paradigm: caller only needs to observe send and returned data whereas the details of the transmission handling are hidden.

II.1 GENERIC TESTBENCH ENVIRONMENT

Each testbench thread communicates with the neighbors. For example, (see the Figure 3.1), the generator passes the stimulus to the agent. The environment class needs to know when the generator completes and then tell the rest of the testbench threads to terminate. This can be achieved with interprocess communication constructs such as the standard Verilog events, event control and wait constructs, and the SystemVerilog mailboxes and semaphores [2].

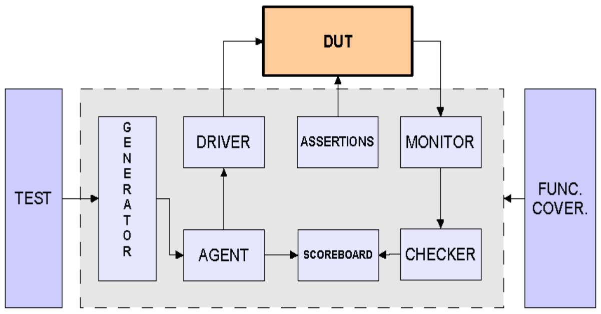


Fig. 3.1. Generic testbench environment

To avoid racing conditions and undetermined behavior caused by non-determinstic order of processes evaluation within simulation, all threads in the testbench should be synchronized to exchange data properly. At the most basic level, one thread waits for another (such as the environment object waiting for the generator to complete). Multiple threads may try concurrently accessing a single shared resource, such as bus in the DUT – the testbench needs to ensure that one and only one thread is granted with write access [2]. At the highest level, threads need to synchronously send each other the data, such as transaction objects that are passed from the generator to the agent

The testbench structures are used in the operational domain of the testbench (environment) to control the operation of the DUT by supplying stimulus and capturing the response [17]. To make the testbench useful its components need to detect the answer for two basic questions (in operational domain): “Does it work?” and “Are we done?”. Such components represent the analysis part of the testbench: scoreboards and coverage collectors. Scoreboards are analysis components whose role is to answer the question “Does it work?”. Scoreboards detect whether the actual DUT response on stimulus match the expectations.. Typically, coverage collectors are analysis components that answer the question “Are we done?”, determining whether the testing has reached the functional coverage goals (they compute coverage based on the stimulus and response traffic [20]).

II.2 TRANSACTOR AS BASIC «BUILDING BRICK»

Pin-to-pint connections level is used in RTL models to communicate with each other. Particular bits are sent via particular wires – the «mediator» between elements is a net. A net can drive a single bit value at single moment of time, and this value may be changed by the driving component. When the value of net is changed, all components that are driven by this net should be re-evaluated to reflect this new change on particular inputs.

A transaction is a single transfer of control or data between two entities [2]. Transaction level models consist of multiple processes communicating with each other by sending transactions back and forward through channels.

At the most basic level, a transactor is simple loop that receives a high-level transaction invocation command from a previous block and performs necessary transformations to translate it to lower pin-to-pin level interface [4] (note, that some transactors – such as «generator» – do not have any parent control blocks – generators simply initiate transactions, whereas others – such as the «driver» – receive a transaction to transform and send it to the downstream transactors as a signal transitions). Following SystemVerilog code represents a basic temaplate for implementation of transactor class:

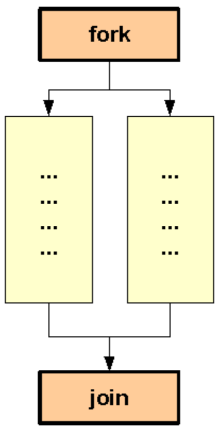
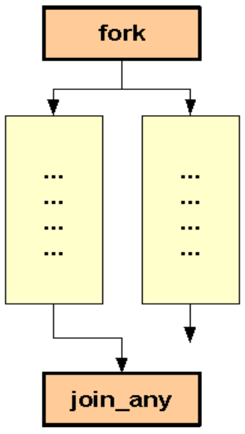
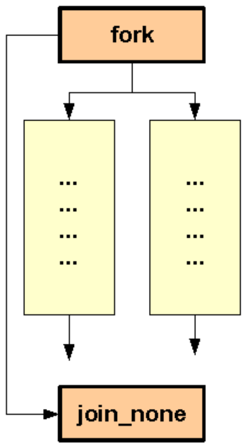
```
class Transactor;
  Transaction TRAN;
  task run;
    forever begin
      // (1) receive the transaction from upstream block
      ...
      // (2) do some processing/transformations
      ...
      // (3) send to downstream block
      ...
    end
  endtask
endclass
```

Most blocks within the testbench environment are running in dedicated parallel threads. Simulation environment maintains a *queue of processes* and schedules thread activations between timing control statements imitating their parallel runtime semantics. [8]; SystemVerilog provides basic mechanisms to run and control different kinds of parallel execution threads [5]:

– a **fork-join** block always causes the process executing the fork statement to block until the termination

of all forked processes (this mechanism is inherited from Verilog [13]);
– with the addition of the **join_any** and **join_none** mechanisms, SystemVerilog provides three choices to control when the parent (forking) process resumes execution (see Table 3.1);

TABLE 3.1
SYSTEMVERILOG MECHANISMS TO SPAWN PARALLEL EXECUTION THREADS

	join	**join_any**	**join_none**
Description	The parent process blocks until all the processes spawned by this fork complete	The parent process blocks until any one of the processes spawned by this fork completes	The parent process continues to execute concurrently with all the processes spawned by the fork
Semantics			

II.3 INTERPROCESS SYNCHRONIZATION: SEMAPHORES

High-level and easy-to-use synchronization and communication mechanism are essential to control the kinds of interactions that occur between dynamic processes used to model a complex system or a highly reactive testbench [9]. Shared resources might become a bottleneck when used by multiple threads – the term «critical section» is introduced to classify the segment of code where a process may be changing shared variables [18]. There are two kind of threads possible: writing threads (that modify the shared data and require mutual exclusion) and reading threads (they don't modify data and could access the shared data simultaneously).

In order to maintain data consistency, there is a need for special scheduling mechanisms to ensure predictable execution order between concurrent processes. Consider the «critical section problem» described below:
– each process must request permission to enter its critical section (order of execution as shown in the following listing)

```
do {
   // entry section (request permission to modify common variables)
   // critical section
   // exit section (typically, release locks)
   // remainder section
} while (true);
```

– the problem is «how to design the entry section and the exit section»
Possible solutions of this problem are the following [7]:
– **mutual exclusion** (if a process is executing its critical section, then no other processes can execute the same critical sections simultaneously);
– **progress** (if critical section is currently free, but there are several requests to enter the critical section, then the selection of the process to enter the critical section cannot be postponed indefinitely);
– **bounded waiting** (there must be a bount on the number of times that other processes attempt to enter the critical sections after request initiation till it is granted);
One of the clear solutions introduced in SystemVeril og is a **semaphore** object, that behaves like a mutually exclusive synchronization tool that does not require explicit waiting (see Figure 3.2 – Mutual exclusion with semaphore).

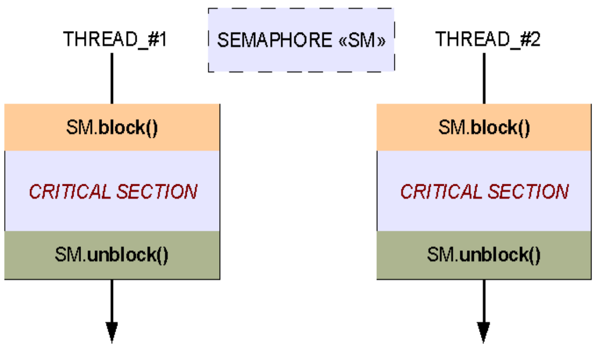


Fig. 3.2 – Mutual exclusion with semaphore

There are following major requirements imposed on the implementation of semaphore [6]:

– each semaphore is associated with its own process requests waiting queue;

– waiting queue is a FIFO of processes, so the one requesting access earlier is granted first.

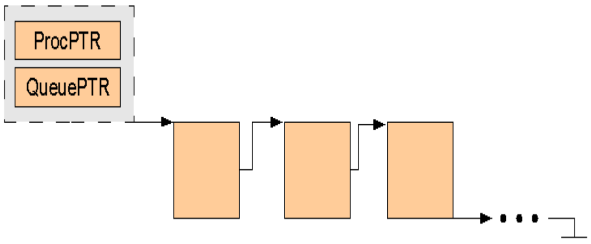


Fig. 3.3 – Waiting queue

– following operations are required:
  – lock (place the process invoking the operation at the end of the queue)
  – resume (drop the first process from the queue and reactivate it))

SystemVerilog standard defines the «semaphore» class that should provide the functionality identical to described above. A thread that requests a key when one is not available always blocks (multiple blocking threads are queued in FIFO order). This mechanism together with threads and the related control constructs complement the dynamic nature of OOD [10]. As objects are created and destroyed, they can run in independent threads, allowing building a powerful and flexible testbench environment.

II.4 INTERPROCESS SYNCHRONIZATION: MAILBOXES

Mailboxes are another convenient way for inter-process communication. Semaphores provide a way for processes to communicate *indirectly*; mailboxes are facilities where processes can communicate *directly* with one another. Mailboxes operate in the same manner as a real mailbox. A process may submit a letter (which, in this case, will contain a data message) for another process in a mailbox. The message stays in the mailbox until the receiving process comes and retrieves it. If the receiving process checks the mailbox before the sending process deposits the message in the mailbox, the receiving process goes back empty handed [10].

Following is the list of required operations [21]:

– create using the object-oriented new() method;

– allow messages of any data type to be delivered and retrieved or to only allow messages of a specific data type to be delivered and retrieved;

– capacity can be unconstrained (any number of messages can be delivered without retrieving messages) or constrained (to have a maximum capacity);

– methods to deliver messages, retrieve messages, and to examine the mailbox contents;

Figure 3.4 illustrates an example of mailbox usage case: it has FIFO queue nature that allows to be used effectively for comparison of expected and actual results.

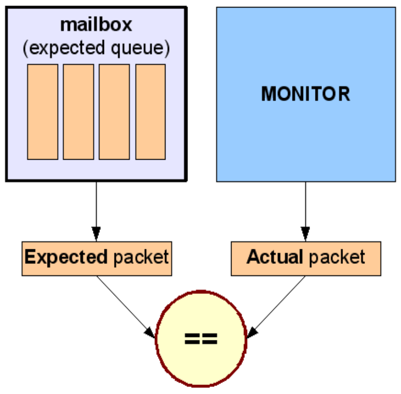


Fig. 3.4 – Using mailbox for checker component

When a letter is delivered and put into the mailbox, one can retrieve the letter (and any data stored within). However, if the letter has not been delivered when one checks the mailbox, one must choose whether to wait for the letter or retrieve the letter on subsequent trips to the mailbox. Mailboxes should provide processes to transfer and retrieve data in a controlled manner. Mailboxes are created as having either a bounded or unbounded queue size. A bounded mailbox becomes full when it contains the bounded number of messages. A process that attempts to place a message into a full mailbox shall be suspended until enough room becomes available in the mailbox queue. Unbounded mailboxes never suspend a thread in a send operation.

III. CONCLUSION

Modern digital designs are extra-complex and they require powerful and agile testbenches. This problem of building complex verification frameworks is significantly simplified with object-oriented approach when whole system is represented as a collection of small components with clearly defined structure and interfaces. The problem is further simplified when these components are reusable. Such components are built through the use of standard TLM interfaces and object oriented programming techniques with semaphores.

The design is modeled as many independent blocks running in parallel and the testbench must also generate multiple stimulus streams and check the responses using parallel threads. These are organized into a layered testbench. SystemVerilog introduces powerful constructs such as fork...join_none and fork...join_any for dynamically creating new threads. These threads communicate and synchronize using events, semaphores, mailboxes, and the classic @ event control and wait statement.

REFERENCES

[1] Adam Rose, Tom Fitzpatrick, Dave Rich, Harry Foster. Advanced Verification Methodology Cookbook, 2.0, July 24, 2006

[2] Chris Spear. Systemverilog for verification, A Guide to Learning the Testbench Language Features. Synopsys, Inc., Springer Science+Business Media, LLC, 2006.

[3] Stuart Sutherland, Simon Davidmann, Peter Flake. SystemVerilog For Design, A Guide to Using SystemVerilog for Hardware Design and Modeling. Springer Science+Business Media, LLC, 2006.

[4] Hardware Verification With SystemVerilog. An Object - Oriented Framework. Mike Min tz, Robert Ekendahl. Springer Science+Business Media, LLC, 2007.
[5] IEEE IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. New York: IEEE 2005.
[6] Episodes on Operating Systems, Peter Sturm, Universit y of Trier, 2007.
[7] Operating System Concepts. Silberschatz, Galvin and Gagne, 2005.
[8] Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale. Verification Methodology Manual for SystemVerilog. Springer, 2005
[9] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vissides, John. Design Patterns: Elements of Reusable Object - Oriented Software. Reading, MA: Addison -Wesley 1995.
[10] Project VeriPage. SystemVerilog Interprocess Synchronization and Communication. http://www.project - veripage.com/ipcs_1.php, 2007
[11] Deepak Kumar Tala. Interprocess Communication. http://www.asic-world.com/ systemverilog/sema_mail_events1.html, 2007
[12] Stuart Sutherland. A Guide to the New Features in the Verilog Hardware Description Language, 2001
[13] IEEE IEEE Standard Verilog Hardware Design, Description Language. New York: IEEE 2001
[14] Synopsys, Inc., Hybrid RTL Formal Verification Ensures Early Detection of Corner-Case Bugs, 2003
[15] Van der Schoot, Hans, and Bergeron, Janick Transaction - Level Functional Coverage in SystemVerilog. San Jose, 2006
[16] Gamma, Erich, Helm, Richard. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1995
[17] Bergeron, Janick. Writing Testbenches Using SystemVerilog. Springer, 2006
[18] Stuart Sutherland, Don Mills. Verilog and System Verilog gotchas – TestBench gotchas. http://www.deepchip.com/items/0466 -07.html, 2007.
[19] William k. Lam. Simulation and Formal Method -Based Approaches. Prentice Hall 2007.
[20] Andrew Piziali, Functional Veri fication Coverage Measurement and Analysis. Springer 2007.
[21] Stuart Sutherland, Modeling FIFO Communication Channels Using SystemVerilog Interfaces. http://www.sutherland-hdl.com/papers/ 2004 – SNUG – Boston – paper_SystemVerilog_FIFO_Channel.pdf
[22] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, Kelli A. Houston. Object-Oriented Analysis and Design with Applications. Addison-Wesley Object Technology Series, 2005.