CONSTRUCTING REUSABLE TESTBENCHES

Alex Wakefield, Synopsys Professional Services, Marlboro, MA
Bassam Jamil Mohd, Synopsys Professional Services, Marlboro, MA

Abstract

This paper explores a novel approach to classifying, enabling and measuring testbench reuse. Six levels of testbench reusability are presented ranging from utility (lowest), to communication, transactor, generate/check, configuration and tests (highest). The layers provide a template for implementation and enable reuse on a layer-by-layer basis. A qualitative measure of reuse is available directly from the layered model while a quantitative measure is available by examining the effort required to build the layers.

Introduction

Reducing the cost, time, and effort associated with testbench development is a goal for many design teams. Verification using testbenches and dynamic simulation is an important part of design validation. Reusing pre-tested components will reduce testbench development time in a manner similar to reusing RTL. Efficient reuse can deliver significant savings in these areas if implemented correctly. Accomplishing this task requires a change in methodology that must be accompanied by a method of measuring the reuse in a quantitative and qualitative manner.

This paper will review a layered architecture often used to implement testbenches. We will classify each layer into different categories that clearly define where the functionality should be placed within the testbench. These layers provide an opportunity for reusing the testbench on a layer or component level.

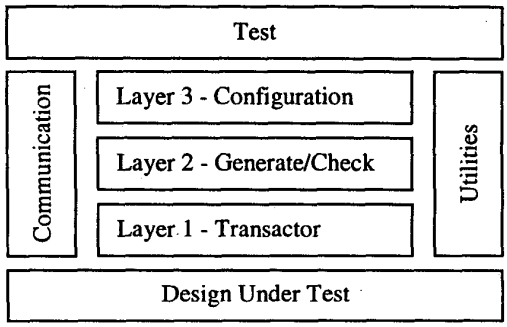
Object-oriented software practices have many advantages directly applicable to the testbench design process. Several of these concepts will be discussed, with particular focus on how they facilitate reuse. We will discuss issues that occur when the problem is partitioned incorrectly.

Finally, we will examine a method for quantifying reuse between designs. Measuring testbench reuse can be performed in several ways, including quantitative measures such as lines of code, and days of effort, or using qualitative measures such as the six levels of reuse explained in this paper. The examples presented and analyses derived are from the authors' experience in packet-based testbench development.

Layered Architecture

A layered methodology has been used to divide and simplify many complex problems into smaller, manageable pieces. This approach is used in the ISO/OSI communications model [1] and is examined in a recent paper by Hawana and Schutten [2]. In our testbench model we propose four horizontal layers and two vertical layers, as shown in Figure 1.

Figure 1. Testbench Architecture



This architecture reduces the testbench complexity by providing a clear focus for the functionality of each layer. Communication between layers is limited as each layer only communicates with the layers directly above and below it. The vertical layers provide communication, and global utility routines. The horizontal layers provide the majority of the testbench functionality, including tests, generation, checking, and stimulating the design under test (DUT).

Higher up in the model, abstraction *increases* and control of the DUT *decreases*. This allows the engineer to write tests efficiently – and, more importantly, allows the tests to be independent of the hardware. This also means a conformance suite of design-independent tests can be created.

Lower down in the model, abstraction *decreases* and control *increases*. The transactor models will have little, if any, knowledge of the test, but a lot of detail about the timing and low-level protocol issues. The transactor does not know if a packet will be dropped by the DUT due to a bad data field or incorrect address.

The layered architecture provides the test writer with level of control similar that of a motor-vehicle driver. To stop a vehicle, a driver presses on the breaks without the knowledge of low-level mechanical details. The driver instead focuses on the task of driving the vehicle. Similarly, a test writer will focus on creating test scenarios to fully verify the DUT functionality and rather than the pin level details.

0-7803-7655-2/02/$17.00©2002 IEEE

If the interface to each layer is clearly defined and implemented in the same manner, then layers will be interchangeable between testbenches. This reuse significantly reduces the effort required to build a new testbench, as a portion of the code is already developed and tested.

Layers

The first step in establishing a common set of layers suitable for multiple testbench designs is to define the functionality of each layer. This definition will clearly specify the focus of each layer and the interface between layers. A definition of each layer is as follows:

Utility

Utilities are often implemented as a set of global functions, including items such as printing and logging. The advantage here is that the functions can be called from anywhere in the code and are easily implemented. The disadvantage, however, is that the functions are very limited in the static information they can contain. Static data will apply to the entire testbench as the function may be called from multiple locations in the testbench, possibly in the same time step. Reuse in many Verilog or C testbenches consists of only global utility functions.

Communication

The communication layer is associated with multiple protocols and provides a standard interface used to connect different components together. This layer contains packet queues to store messages that can be an events or data structures. Any testbench component is able to send or receive messages using the communication layer. Complex communication layers contain advanced notification schemes to synchronize testbench components.

The implementation of the communication layer differs based on the verification language. The layer may consist of signals in a VHDL testbench, wires in a Verilog testbench, mailboxes in Vera [3], or channel classes in SystemC [4]. The functionality may be built into the language or be custom classes written for specific behavior – for example a packet class. In Vera, a mailbox does not have a type and accepts any object or type as input and output. This provides a simple, robust implementation in instances where a mailbox is used for a single protocol.

Transactor

Transactors are associated with a specific protocol and are typically reused in any testbench utilizing that protocol. The transactor is the interface between the testbench and the DUT. Traditional bus-functional models (BFM) are one type of transactor – for example sending packets or executing read/write commands on a bus. The transactor has two main parts: a transmitter and receiver. The transmitter transforms data from the testbench onto the DUT interface. A transmitter is a function called from the testbench or a thread that blocks when reading from the testbench. The receiver is a thread that continually samples the DUT, accepts packets and sends the packets to the check layer.

Generate/Check

A generator is often associated with a single protocol. The generator creates random data units (packets or commands), and sends a copy of the data to both the transmitter and the predictor. The test controls randomization through settings on the generator. By changing the constraints, the generator will create different streams of valid or error-producing data.

A predictor is associated with the source protocol, destination protocol, and transfer function between the protocols, mirroring the DUT behavior. The predictor receives a copy of the stimulus data and produces an expected result. Predictor functionality can range from a simple queue to a complex instruction-set processor model.

A checker is often associated with a single protocol. The checker receives a data unit from the receiver and the predictor. The two items are compared and an error occurs if they do not match. If out-of-order processing is allowed by the DUT, the checker or predictor will need to account for this. The checker may collect statistics including latency and the number of packets received or dropped.

Configuration

The configuration layer has two functions – configuring the testbench and providing a stable interface between the tests and the lower layers. This layer determines the number of transactors, generators, and checkers required and connects the testbench to the DUT. All control from the tests to the lower layers occurs via the configuration layer. This isolates any future changes in the lower testbench layers to a single layer or interface, rather than effecting a potentially large number of tests.

Tests

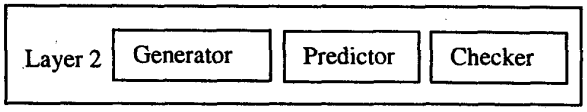
The test code communicates with the testbench by making calls to the configuration layer. Ideally, the configuration layer has powerful controls, allowing the tests to be small and therefore created with minimal effort. The tests should be independent of the lower layers, preferably containing randomization constraints and parameter settings applied via the configuration layer.

Components

Several considerations must be taken into account when implementing layered architecture. Layers should be reusable between different testbenches, extensible to allow new functionality, and be able to share code effectively. One step in achieving these goals is to split each layer into several components, as shown in Figure 2. This limits the

functionality of each component, reduces its complexity and increases potential reuse.

Figure 2. Components



To allow components to be swapped between testbenches, (and for the configuration object to treat all components in a uniform manner), the interface for each component must be clearly defined. All components of the same type should ideally have identical interfaces so that a component from one testbench can be easily reused in another testbench.

Components must be robust, thoroughly tested and easily extendible for design-specific additions. Testbenches for different designs must share functionality in a controlled manner to avoid code duplication.

Objects

Object-oriented languages such as Vera, C++ and SystemC are well suited to implement components using classes and objects. They provide an enforceable interface and a controlled code-sharing mechanism.

A class separates the interface and implementation into two well-defined areas: the class header and the class body. The class header contains data members and function declarations while the class body contains function definitions. This allows the class designer to strictly control the interface via the class header.

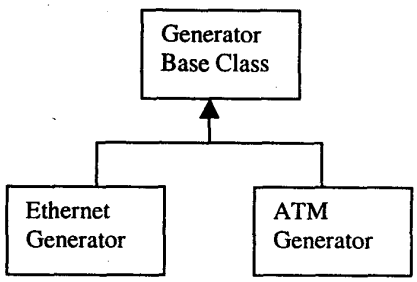
Deriving all objects of the same component type from a single base class will ensure that the interfaces for all objects are identical. Components can then be swapped between testbenches with minimal effort. Care must be taken to ensure a base class interface is suitable for most situations, but is not so complex as to make the class unusable. Derived objects have access to the parent data and methods, and can also introduce additional data and methods.

Base Classes

Objects control code reuse by collecting common elements into a base class and deriving implementation classes, as shown in Figure 3. Derived classes extend the functionality for a particular component, forming a small hierarchy of classes.

A base class can be defined to contain functionality common to all components of that type. As all code should exist in one location only, duplicated code should be collected as far up in the inheritance hierarchy as practical. Any classes derived from this class can use the methods, data elements and shared code. This will eliminate duplicated code and reduce maintenance. Objects can successfully and efficiently implement each component using a hierarchy of classes. Virtual methods complete the framework and allow a function defined in multiple classes to be called using a base-class reference. Different component types do not share functionality, so multiple inheritance is not required.

Figure 3. Base and Derived Classes



Layer Communication

The type of tests needed to stimulate the hardware will influence the design of the testbench components. Valid data tests will often interact solely with the generator component specifying traffic-shaping parameters. Error injection and flow control tests will need to communicate with the lower layers via the communication layer. This can be accomplished through two methods: either function calls to the lower layers, or by adding data members to the packet classes.

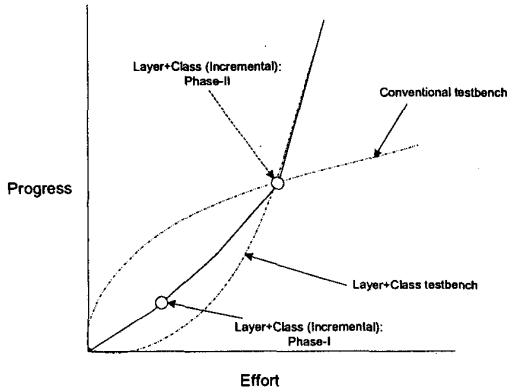
Function calls from the tests to the lower layers limits test reuse, as the tests are now dependent on the lower layers. This is further complicated as a single change to the lower layers may impact a large number of tests. The configuration layer helps isolates these changes, however care must be taken to ensure that the configuration object does not become overly complex.

Adding data members to the packet class for communication down the layer stack removes the need for direct function calls. Deriving error classes for each component and adding error data fields into these new classes manages the additional complexity. The original generator, transactor and checker classes will continue to function as they use the non-error packet data class. The error tests and classes are created and debugged independently in an isolated environment [3]. Careful class design must be used to control the number of derived classes and maintain a consistent interface between multiple error classes.

Development

The implementation of the proposed testbench (using layers and classes) has a slower start compared with the conventional (non-layer, non-class) testbench as shown in Figure 4. Once the classes have been implemented and integrated, the proposed testbench verification progresses at a faster rate.

Figure 4. Testbench implemtation comparison



If the DUT is available and ready for verification at the start of the project the initial slow pace can be frustrating to the verification and design teams. One solution is to incrementally release the layers (for example, using two phases as show in Figure 4.) During the development of the testbench, each layer can be incrementally improved. Initially, simulation can begin using thin layers and a basic testbench. The thin layers can be extremely simple to start with – for example replace a generator with a function call containing fixed arguments, or replace a checker with a print statement. The layers can then be progressively developed and tested.

As the classes are reused, the code needs to be maintained, and of higher quality than is typical in single-use testbenches. This improved quality has a cost, however this cost is lower than if the code was recreated for each testbench.

Quantitative Analysis

This section compares the effort required to build a testbench with and without base classes. The effort required to construct a testbench includes developing classes, integrating objects and writing tests, as shown in equation (1).

Effort Without Base Classes=
develop utility-layer classes
+ develop communication-layer classes
+ develop transactor-layer classes
+ develop generate/check-layer classes
+ develop configuration-layer classes
+ testbench integration
+ writing tests (1)

Developing a class involves the creation of an interface and its functions. With careful planning, the functionality can be split into two groups – basic functionality common to several components, and complex functionality specific to a single implementation. The interface and basic functions form the base class, while the complex functions form the derived class. The effort required to build each of these is shown in equations (2) and (3).

Base Class Effort =
develop interface
+ develop class basic methods (2)

Derived Class Effort =
develop class complex methods (3)

Once the base classes are developed and debugged, constructing classes in the new testbench becomes a matter of developing the derived classes. Also, testbench integration stated in equation (1) is almost eliminated since the integration issues have been resolved in the class interface. Therefore the effort required to construct a new testbench using the base classes is given by equation (4).

Effort with base classes =
develop utility-layer derived classes
+ develop communication-layer derived classes
+ develop transactor-layer derived classes
+ develop generate/check-layer derived classes
+ develop configuration-layer derived classes
+ writing new tests (4)

The effort required to develop the new testbench can be expressed using the reusability factor per testbench layer as defined in equation (5).

$R_{For\_TB\_Layer}$ = (Effort with reusable component) / (Effort without reusable component) (5)

Hence, equation (4) can be rewritten as shown below in equation (6).

Effort to construct testbench using base classes =
(develop utility-layer classes) * (1- R_{UTIL})
+ (develop communic-layer classes) * (1- R_{COMM})
+ (develop transactor-layer classes) * (1- R_{TRANS})
+ (develop gen/check-layer classes) * (1- $R_{GEN/CHK}$)
+ (develop config-layer class) * (1- R_{CONFIG})
+ (writing tests) * (1- R_{TEST}) (6)

When implementing the base classes, the reuse at upper layers implies reuse of the lower layers. Hence, the proposed six reuse-levels can be expressed in terms of layer-reusability factors as shown in Table 1.

Table 1 Levels of Reuse

	Layer Reuse Factors					
Reuse Level	R_{UTIL}	R_{COM}	R_{TRAN}	R_{GEN}	R_{CFG}	R_{TEST}
No Reuse	0	0	0	0	0	0
Utility	> 0	0	0	0	0	0
Comm	> 0	> 0	0	0	0	0
Transactor	> 0	> 0	> 0	0	0	0
Check/Gen	> 0	> 0	> 0	> 0	0	0
Config	> 0	> 0	> 0	> 0	> 0	0
Tests	> 0	> 0	> 0	> 0	> 0	> 0

For example we can apply the analysis to a testbench with following assumptions:

- The testbench uses the layered architecture,
- The DUT is a packet processor with 4 ports,
- Each port has a different bus protocol,
- No cell re-ordering, No-merging

Based on past project experience, we can use the following assumptions to estimate the effort required to construct a testbench. If the effort required to develop a transactor class is x, then the effort required to develop other testbench components is as shown in Table 2.

Table 2 – Effort to develop a testbench

Class	Effort
Utility classes	x
Transactor classes	$4x$
Check/Gen classes	$5x$
Configuration classes	x
Communication classes	$2x$
Writing Tests	$4x$
Integration	x
Total	**18*x***

Also, we will assume that the reusability factor for all the layers is 50%. Table 3 shows the estimated effort required for four different levels of reuse.

Table 3 – Testbench development effort with reuse

	Reuse Level			
Class	**None**	**Trans**	**Config**	**Test**
Utility	x	$0.5x$	$0.5x$	$0.5x$
Communication	$4x$	$2x$	$2x$	$2x$
Transactor	$5x$	$2.5x$	$2.5x$	$2.5x$
Check/Gen	x	x	$0.5x$	$0.5x$
Configuration	$2x$	$2x$	x	x
Tests	$4x$	$4x$	$4x$	$2x$
Integration	x	x	x	x
Total Effort	**18*x***	**13*x***	**11.5*x***	**9.5*x***

Conclusion

Constructing reusable testbenches is a complex task requiring considerable planning and foresight. A layered architecture divides this complex task into several manageable pieces enabling reuse at a layer or component level. It is important to control the layer interface and code-sharing methodology through classes.

Classifying each component into six categories allows us to clearly focus on and quantify the reuse. Using the proposed base classes increases reuse and reduces the effort required to construct new testbenches. Reuse can be *qualitatively* measured by examining how many levels of the layered model are shared. Reuse can be *quantitatively* measured by applying the reuse-factors presented in this paper.

References

1. Tanenbaum, Andrew S, "Computer Networks", Prentice-Hall International Editions, 1988, pp 14.1.
2. Hawana, Mohammed and Schutten, Rindert, "Testbench Design, A Systematic Approach", Design Verification Europe Conference, 2001.
3. Haque, Faisal I. and Khan, Khizar A. and Michelson, Jonathan, "The Art of Verification with Vera", Verification Central, 2001, pp 153 – 160, pp. 343 – 347.
4. Grötker, Thorsten and Liao, Stan and Martin, Grant and Swan, Stuart, "System Design with SystemC", Kluwer Academic Publishers, 2002, pp 87 – 130.