Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches

David Rich
Mentor Graphics, Inc
San Jose, CA
dave_rich@mentor.com

Jonathan Bromley
Doulos Ltd
Ringwood, England
jonathan.bromley@doulos.com

***Abstract*— Sophisticated functional verification environments using SystemVerilog typically make use of the language's object-oriented programming features to build a flexible, re-usable test environment and components that can easily be extended and reconfigured. Ultimately, though, the test environment must interact with the signals, events, and clock cycles of the device-under-test (DUT) and its supporting structures. Published tutorial and methodology material on SystemVerilog has overwhelmingly recommended use of the virtual interface construct to achieve this interaction. A virtual interface is a reference to a static interface instance. The class-based test environment, constructed dynamically at the beginning of a simulation run, can interact with a DUT through such virtual interfaces, allowing it to be written without knowledge of the detailed static instance hierarchy of the DUT and its surroundings. In such a test environment, manipulation of DUT signals is performed by a bus functional model (BFM) object that makes reference to signals in the DUT only through a virtual interface that will be bound to an interface instance at run time.**

This paper reviews this style of using virtual interfaces, and then presents in detail an alternative approach in which the BFM is not implemented in the class-based SystemVerilog testbench. Instead, a suitable generic BFM abstract base class is created. Virtual methods in this base class provide access to all the BFM functionality that will be required by a testbench, but no implementation is provided. Instead, the concrete BFM is implemented as a class derived from this generic BFM, but whose code is embedded in the SystemVerilog module or interface that will be instanced alongside the DUT to connect to its signals. We show that this approach brings a number of practical and methodological benefits, minimizing the testbench code's dependence on details of the DUT's connections and providing a more intuitive separation of concerns at the lowest level of the testbench while fitting neatly into an object-oriented testbench design style.

Finally this paper will consider how this style of BFM and the more conventional style based on virtual interfaces interact with other SystemVerilog constructs, such as clocking blocks, to further isolate the testbench from the design.

I. THE TWO KINGDOMS OF THE VERIFICATION WORLD

For the current authors, and for the majority of working verification engineers in our industry, a design-under-test or device-under-test (DUT) is most likely to be coded in Verilog or VHDL using the register-transfer level (RTL) of abstraction. At this level, information about the current state of the design is carried on signals (nets or variables, in Verilog) and the design advances from its current state to a future state thanks to signal updates executed in response to signal value changes. The design's topology is determined once and for all at the start of simulation, in a process generally known as *elaboration* of the design.

By contrast, the test environment that is used to exercise a DUT in simulation may be constructed in a variety of styles. However, recent developments in languages and tools for verification have encouraged the adoption of object-oriented programming (OOP) techniques for building the test environment. An environment built using OOP has a topology that is determined not in a distinct elaboration step, but instead by the action of procedural software that constructs verification objects and assembles them into a complete testbench. Although it is common (and often very convenient) for this testbench topology to be constructed early in the simulation, and to be left unmodified thenceforward, it is certainly possible in principle for the testbench topology to be modified dynamically during the course of a simulation run.

In consequence, we find that typical verification environments naturally fall into two sections: on the one hand the statically-instantiated invariant topology of RTL DUT code and perhaps some support structures for the DUT; and on the other hand, the dynamically-constructed and flexible structure of the OOP testbench. The coding styles, design approaches and dominant concerns of these two sections are so different that we are tempted to think of them as separate kingdoms in the verification world. But they are kingdoms that must of necessity share a border. It is the frontier-posts of that border that concern us in this paper.

II. THE CHALLENGE OF CONNECTING OBJECTS TO RTL

SystemVerilog [1] offers object-oriented programming within a language that also fully supports all existing constructs of the Verilog Hardware Description Language(HDL) [2]. Consequently, verification code written using SystemVerilog's object-oriented features can directly read and manipulate nets and variables in an RTL design using Verilog's hierarchical name resolution mechanism. Figure 1 shows how this could be achieved, using a trivial example in which the RTL design contains only one variable, and that variable is stimulated by code in the `run()` method of a SystemVerilog class `Stimulus`. Note that the stimulus generator object (instance `stimulus` of class `Stimulus`) is constructed dynamically by code in the testbench module `stimulus_module`.

Figure 1: Trivial RTL DUT stimulated by a SV object

```
module dummy_RTL_module(input R);
  ...
endmodule

package stimulus_pkg;
  class Stimgen;
    task run();
      repeat (10)
        #5 testbench_Top.R = 1'b0;
        #5 testbench_Top.R = 1'b1;
      end
    endtask
  endclass
endpackage

module stimulus_module;
  import stimulus_pkg::*;
  initial begin
    Stimgen stimgen;
    stimgen = new;
    stimgen.run();
  end
endmodule

module testbench_Top;
  reg R;
  dummy_RTL_module DUT(R);
  stimulus_module  tester();
endmodule
```

While this mechanism clearly works, and is straightforward, it fails to address a number of important practical concerns for verification environment developers. Some of the most pressing of these concerns are described below.

A. *Synchronous Signal Timing Abstraction*

In a typical modern design, large groups of signals are likely to be synchronous to a clock (although, of course, there may be many such clock domains in a big design). Within each clock domain, all synchronous signals should be driven and/or sampled with some fixed timing relationship relative to the clock. It is straightforward, but tedious, to manage these timing relationships using procedural code in the testbench. Such timing clutters the procedural code and confuses two levels of abstraction that most verification engineers prefer to keep distinct: the event-driven, signal-level abstraction of the RTL code, and the cycle-based abstraction of the testbench in which timing is expressed in clock cycles rather than nanoseconds.

B. *Encapsulation and Re-use of Verification Code Elements*

In good object-oriented programming practice, classes form self-contained, re-usable, extensible elements. The `Stimulus` class in Figure 1 clearly does not meet these criteria. It contains not merely a hard-coded signal name but even a hard-coded Verilog hierarchical path name. Consequently it would be useless in an even slightly modified verification environment.

Stimulus and monitoring elements are very likely to be written to interact with a standardized set of signals or interface protocol, and therefore have the potential for wide re-use across verification projects. Any element containing hard-coded path or signal names is immediately disqualified from such re-use.

C. Clumsy Code

The need to specify signal names with full hierarchical qualification on each occasion the signal is modified leads to code that is difficult to read and difficult to maintain. It is clear that some indirection is required between the testbench class and the RTL with which it interacts.

III. VIRTUAL INTERFACE

The indirection mentioned above requires the testbench class to have some kind of reference or pointer into the RTL environment. SystemVerilog provides a specific mechanism to achieve this, known as a *virtual interface*.

Before discussing virtual interfaces, we will first describe briefly the *interface* construct of SystemVerilog.

A. Interface

An *interface* is a design unit in SystemVerilog, broadly similar to a module, but having certain special features that make it especially well suited to the description of interconnect structures. Interfaces are described in more detail in references [3], [5] and [7]. Interfaces are like modules in that they are statically instantiated, with their instances becoming members of the elaborated hierarchy. Consequently, any interface instance has a Verilog hierarchical path name.

B. Taking a Reference to an Interface

A *virtual interface* is a SystemVerilog variable that can hold a reference to an interface instance. A variable of virtual interface type can be given a value (i.e. can be made to reference an existing interface instance) by assigning the hierarchical path name of the chosen interface instance to it. Once this has been done, members (nets and variables) of the referenced interface instance can be accessed using the "`.`" select operator, just as if the variable stood for the interface's full hierarchical path name.

In other respects, though, a virtual interface variable is like any other variable in that it can be passed as an argument to a subprogram, copied to another variable of appropriate type, compared for equality with another variable and so forth.

Figure 2 modifies the code example of Figure 1 to use an interface to contain the signal that will be manipulated by the testbench, with a virtual interface holding a reference to the appropriate interface instance. The organization of this code is now much more satisfactory than that of Figure 1. The interface declaration `dummy_intf` encapsulates all the connections needed to hook to the DUT module, or any module having a similar external interface. The testbench class `Stimulus` no longer needs any knowledge of the layout of the DUT and its surrounding infrastructure. Instead it merely knows that it is interacting with *some* instance of a `dummy_intf`, which is known to contain a well-understood collection of nets and variables representing the interconnect structure of interest. A reference to that instance will be passed into the object via its constructor argument - and, indeed, could be updated later in the life of the simulation if required.

Figure 2: The example of Fig.1 modified to use a virtual interface

```
module dummy_RTL_module(input R);
  ...
endmodule

interface dummy_intf();
  reg R;
endinterface

package stimulus_pkg;
  class Stimgen;
    virtual dummy_intf V;
    function new(virtual dummy_intf V);
      this.V = V;
    endfunction
    task run();
      repeat (10)
        #5 V.R = 1'b0;
        #5 V.R = 1'b1;
      end
    endtask
  endclass
endpackage

module stimulus_module;
  import stimulus_pkg::*;
  initial begin
    Stimgen stimgen;
    stimgen = new(testbench_Top.di);
    stimgen.run();
  end
endmodule

module testbench_Top;
  dummy_intf di();
  dummy_RTL_module DUT(di.R);
  stimulus_module  tester();
endmodule
```

C. Virtual Interfaces in a Wider Context

For use in a realistic, generally applicable verification methodology there remain some unsatisfactory features in Figure 2. Although the stimulus class is now usefully decoupled from the DUT and test environment, and therefore can be re-used, it remains tightly coupled to the `dummy_intf` interface definition; these two pieces of code therefore must remain locked together. It is also necessary to hook the DUT to the signals in this interface. In our example we have chosen to do it by hierarchical reference into the interface instance, but other methods are possible (see for example [7]).

Finally, we note that it is necessary to pass the appropriate instance name as an argument to the stimulus class's constructor, but typically the constructor call is not located in the same piece of code that instantiates the required interface. In the specific example of Figure 2 it is the top-level testbench structure module `testbench_Top` that instantiates the interface, and therefore controls its instance name; but it is the verification component, in module `stimulus_module`, that constructs the stimulus object and therefore needs to know the interface's instance name. This issue is part of the wider problem of verification component configuration, which will be discussed in more detail later.

None of these problems detract from the usefulness of virtual interfaces in providing a flexible connection between the dynamic world of software-like verification components and the statically-instantiated environment of the DUT and its support structures. However, it is clear that careful attention to testbench organization is needed in order to maintain the reusability of class-based verification components.

IV. BUS FUNCTIONAL MODELS

A. *Review of traditional module-based BFMs*

In traditional HDL-based verification practice, a *bus functional model* (BFM) is a component, typically packaged as a module, whose purpose is to mimic the activity found on a collection of signals (such as a bus) without necessarily mimicking the detailed internal behavior of any specific device connected to those signals. Verilog HDL makes it very easy and convenient to create BFM modules. The signals that will be manipulated by the BFM appear as ports of the module, and operations that the BFM can perform are coded as tasks or functions in the module, which will be called by procedural code in a test environment. Figure 3 shows a simple Verilog BFM that imitates the behavior of an asynchronous serial transmitter. Its only port is the simulated serial line output. Task calls into the module can configure the BFM for serial data rate and other features; the necessary configuration information is stored in variables in the module. Another task, `send`, can be called by the test code to cause the BFM to perform appropriate activity on its output port.

B. *BFMs in an OOP verification environment*

Module-based BFMS as described in IV.A are useful in simple block-level testbenches, but the statically instantiated nature of Verilog modules makes them insufficiently flexible for re-use in large verification environments following a modern OOP methodology. Instead we prefer to code our BFMs as classes. Configuration and stimulus-generation tasks are readily modeled as public methods of the BFM class. Verification environments of arbitrary complexity and flexibility can be constructed dynamically, using procedural code in a testbench environment class to construct and manage appropriate instances of the BFM and other classes.

Figure 3: UART transmitter BFM

```
module UART_Tx(output logic line);

  int NBits;
  time BitPeriod;

task setNBits(input int N);
    if (N>0 && N<=10) NBits = N;
  endtask : setNBits

  task setBitPeriod(input time T);
    if (T>0) BitPeriod = T;
  endtask : setBitPeriod

  task send(input logic [9:0] d);
    line = 0;  // start bit
    repeat (NBits) begin
      #BitPeriod line = d[0];
      d = d >> 1;
    end
    #BitPeriod line = 1;  //stop
    #BitPeriod;
  endtask : send

endmodule : UART_Tx
```

Published verification methodology guidelines such as [6], [7], and [9] describe in detail how such environments can be deployed in practice.

References [7] and [9] are somewhat dogmatic concerning the use of interfaces and virtual interfaces to connect class-based BFMs to the statically instantiated DUT structures. Other methodologies are less specific in their recommendations, allowing the environment developers some discretion in implementing this low-level but essential part of testbench functionality. In the remaining sections we describe an alternative style of connection between BFM and DUT using a well-known software engineering design pattern that we believe offers significant practical benefits for re-use and testbench organization.

V. ABSTRACT BFM

A. *Abstract base class to represent the API*

Our starting-point for an alternative style of BFM connection is to note that the public interface (*application programming interface* or API) presented to a verification environment by a re-usable BFM should take the form of a set of virtual methods. In this way, users of the BFM can remain ignorant of its implementation details, and class extension can easily be used to create alternative BFM implementations that nevertheless present identical APIs to the rest of the verification environment.

Such an API can usefully be encapsulated as an *abstract base class* whose sole contents are the public methods, coded as pure virtual methods. Once such an abstract base class has been defined, variables of that class type can be used to hold references to an instance of any concrete BFM class that inherits from the abstract base class. Testbench code that makes use of such a BFM can do so via variables of the

abstract base class type, conveniently decoupling the remainder of the test environment from details of BFM implementation. In software engineering circles, this technique is effectively known as the *template method* design pattern [13]. From the perspective of this paper, however, the most important aspect of this decoupling is that the *connection* of a BFM to its target HDL signals can also be hidden by this same mechanism.

B. Concrete Implementations Tightly Coupled to the DUT

It is preferable to decouple an abstract BFM base class from the DUT and its supporting structures as completely as possible in the interests of reusability. However, there must also be at least one class derived from this base class that implements the BFM's concrete functionality. This implementation class must manipulate the DUT signals directly, and it is convenient for its code to exist in a scope where those signals are directly visible. This scope should be the module or interface that implements the set of signals that the BFM will manipulate.

Figure 4 presents an example of an abstract BFM class `APB3_BFM` forming the public API to a BFM that can perform transactions on the well-known APB3 peripheral bus structure [10]. The base class is declared in a common package will be imported into a number of different scopes. Note how the base class is specified to be abstract by means of the `virtual` qualifier in its declaration. The interface `APB3_TB_intf` creates the standard set of APB3 bus signals, and also declares a derived BFM class `APB3_concrete_BFM` that implements the API declared in the base class. Within the same interface, an instance of the concrete BFM class is created and initialized. A reference to this instance can then be passed to any part of the testbench that has a variable of `APB3_BFM` type, and calls to the concrete class's methods can be made through that variable.

VI. Benefits of This Structure

The arrangement outlined in V. has a number of useful benefits. Because the concrete BFM implementation class is embedded in the same scope that declares the relevant bus signals, it has direct access to those signals without any form of hierarchical reference. Furthermore, the BFM code is naturally locked to the bus signal declarations, avoiding the need to keep two independent design units together. The package `APB3_pkg` that defines our abstract BFM class can readily be imported into any design unit that needs it, without concern for the implementation details of the bus or its BFM.

The bus signals are declared in an interface. This interface can easily be extended to incorporate modports [1][5], providing a convenient way to connect the interface to SystemVerilog RTL designs that would normally be connected to a physical bus interface.

Figure 4: Abstract BFM class with concrete implementation in an interface

```
package APB3_pkg_Fig4;
  virtual class APB3_BFM;
    pure virtual task write (
        int unsigned addr, data);
    pure virtual task read (
        int unsigned addr,
        output int unsigned data);
  endclass
endpackage
```

```
interface APB3_TB_intf(input bit PCLK);
  import APB3_pkg_Fig4::*;
  logic PENABLE, PWRITE, PSEL, PREADY;
  logic [15:0] PADDR, PWDATA, PRDATA;
  class APB3_concrete_BFM extends APB3_BFM;
    task write(int unsigned addr, data);
      @(posedge PCLK)
        PSEL <= 1'b1;
        PENABLE <= 1'b0;
        PWRITE <= 1'b1;
        PADDR <= addr;
        PWDATA <= data;
      @(posedge PCLK)
        PENABLE <= 1'b1;
      do @(posedge PCLK); while (!PREADY);
        PSEL <= 1'b0;
        PENABLE <= 1'b0 ;
    endtask
    task read ( int unsigned addr,
                output int unsigned data);
      @(posedge PCLK)
        PSEL <= 1'b1;
        PENABLE <= 1'b0;
        PWRITE <= 1'b0;
        PADDR <= addr;
      @(posedge PCLK)
        PENABLE <= 1'b1;
      do @(posedge PCLK); while (!PREADY);
        PSEL <= 1'b0;
        PENABLE <= 1'b0 ;
        data = PRDATA;
    endtask
  endclass
  APB3_concrete_BFM bfm = new;
endinterface
```

```
module APB3_TB_top;
  import APB3_pkg_Fig4::*;
  bit CLK;
  always #5 CLK = ~CLK;  // clock generator
  APB3_TB_intf apb3_intf(CLK);
  APB3_device DUT(
    .PCLK(CLK),
    .PENABLE(apb3_intf.PENABLE),
    ...);
  initial begin : Test_Activity
    APB3_BFM bfm;  // abstract BFM reference
    int read_data;
    bfm = apb3_intf.bfm;  // reference to BFM
    bfm.write(100, 1234);
    bfm.read(100, read_data);
    if (read_data != 1234)
      $display("error: unexpected read data");
  end
endmodule
```

VII. USING THIS STRUCTURE WITH VIRTUAL INTERFACES

As shown in Figure 4, the abstract BFM mechanism makes virtual interfaces unnecessary for accessing the BFM. However, virtual interfaces may nevertheless be useful in providing an easy way for class-based verification components to locate the concrete BFM variable in the interface instance.

VIII. CLOCKING BLOCKS FOR TIMING ABSTRACTION

Within a SystemVerilog module or interface, any collection of signals (nets or variables) may be grouped according to the clock signal that normally controls their updating or sampling, using the *clocking block* construct. A clocking block insulates a class-based verification environment from the nanosecond-by-nanosecond minutiae of signal transition timing and allows the testbench to operate in terms of clock cycles. It also relieves the testbench of the need to understand the differences between nets and variables at the interface to the DUT; all signals controlled through a clocking block appear to the testbench as a special kind of variable known as a *clockvar*. Tutorial review of the clocking block mechanism may be found in [4] and [8].

Figure 5 shows the code of Figure 4 reworked to use a clocking block in the APB3 bus interface. It is interesting to note that the concrete BFM class definition is embedded

Figure 5: Abstract BFM class and concrete implementation in an interface using clocking block

```
package APB3_pkg;
  virtual class APB3_BFM;
    pure virtual task write (
        int unsigned addr, data);
    pure virtual task read (
        int unsigned addr,
        output int unsigned data);
    pure virtual task init ();
    pure virtual task idle (int unsigned cycles);
  endclass
endpackage
```

```
module APB3_TB_top;
  import APB3_pkg::*;

  bit CLK;
  always #5 CLK = ~CLK;  // clock generator

  APB3_TB_intf apb3_intf(CLK);

  APB3_device DUT(
     .PCLK(CLK),
     .PENABLE(apb3_intf.PENABLE),
      ...);

  initial begin : Test_Activity
    APB3_BFM bfm;  // abstract BFM reference
    int read_data;
    bfm = apb3_intf.bfm;  // reference to BFM
    bfm.init;
    bfm.idle(5);
    bfm.write(100, 1234);
    bfm.read(100, read_data);
    if (read_data != 1234)
      $display("error: unexpected read data");
    bfm.idle(100); $stop;
  end
endmodule
```

```
interface APB3_TB_intf(input bit PCLK);
  import APB3_pkg::*;
  logic PENABLE, PWRITE, PSEL, PREADY;
  logic [15:0] PADDR, PWDATA, PRDATA;

  default clocking apb_ck @(posedge PCLK);
    output PENABLE, PWRITE, PSEL, PADDR, PWDATA;
    input PREADY, PRDATA;
  endclocking

  class APB3_concrete_BFM extends APB3_BFM;
    task init;
      apb_ck.PSEL <= 1'b0;
      apb_ck.PENABLE <= 1'b0;
      apb_ck.PWRITE <= 1'b0;
    endtask
    task write(int unsigned addr, data);
      ##0
        apb_ck.PSEL <= 1'b1;
        apb_ck.PENABLE <= 1'b0;
        apb_ck.PWRITE <= 1'b1;
        apb_ck.PADDR <= addr;
        apb_ck.PWDATA <= data;
      ##1
        apb_ck.PENABLE <= 1'b1;
      do ##1; while (!apb_ck.PREADY);
        apb_ck.PSEL <= 1'b0;
        apb_ck.PENABLE <= 1'b0 ;
    endtask
    task read( int unsigned addr,
               output int unsigned data);
      ##0
        apb_ck.PSEL <= 1'b1;
        apb_ck.PENABLE <= 1'b0;
        apb_ck.PWRITE <= 1'b0;
        apb_ck.PADDR <= addr;
      ##1
        apb_ck.PENABLE <= 1'b1;
      do ##1; while (!apb_ck.PREADY);
        apb_ck.PSEL <= 1'b0;
        apb_ck.PENABLE <= 1'b0 ;
        data = apb_ck.PRDATA;
    endtask
    task idle(int unsigned cycles);
      ##(cycles);
    endtask
  endclass

  APB3_concrete_BFM bfm = new;

endinterface
```

within the same interface that declares the clocking block, and therefore can take advantage of the convenient `##n` notation for cycle delays that becomes available thanks to the `default clocking` specification in that interface. This makes it easy to add to the BFM a new method `idle()` that simply idles the bus, with no activity, for a specified number of clock cycles. We have also taken advantage of the `##0` feature of clocking blocks to synchronize task execution with the clock, without unnecessarily wasting a clock cycle, as described in[11]. This form of cycle delay causes procedural code to wait until the next clock cycle unless simulation has already reached exactly the moment of a clock event, in which case `##0` does not wait. This `##0` feature is scheduled for inclusion in the 2008 revision of the SystemVerilog standard, but has already been implemented in at least one commercially available simulator [12].

IX. Conclusions

The style of interaction between testbench and DUT illustrated in this paper offers a number of attractive features:

- Its packaging is convenient and easy to manage. The concrete BFM and the interface or module that it connects to are in the same design unit.
- The concrete BFM has easy access to all required signals, while the abstract BFM provides an appropriate level of abstraction for use by the testbench.
- A clocking block (if used) can easily be located in the right place, in the same module or interface as the signals it controls. This makes coding the concrete BFM more straightforward because it has a `default clocking` in the correct scope, and it ensures that timing concerns don't leak into OO testbench. code
- The abstract BFM base class is a good match with re-usable OO testbench component methodology, and could even be applied to much more general BFMs applicable to many variants of a bus structure in which the same base class and API gave access to widely different concrete BFMs. In this way, the OOP testbench could be yet further decoupled from details of the physical interface to the DUT.
- It is easy to fit into a transaction-level modeling (TLM) testbench structure. The abstract BFM can easily be given a TLM interface instead of the procedural API outlined in this paper.
- Reuse of legacy Verilog BFM code is straightforward. It is only necessary to add a derived-class wrapper to the legacy BFM tasks, which can remain packaged in a module as before.
- This scheme has low impact on published testbench architecture methodologies, and can be retrofitted into them without difficulty.

References

[1] "IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2005, 2005

[2] "IEEE Std 1364 -2005 IEEE Standard for Verilog Hardware Description Language," IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) , 2006

[3] Sutherland, Stuart, S. Davidman, and P. Flake, SystemVerilog for Designers. 2nd ed. Norwell MA: Springer Inc., 2006.

[4] Spear, Chris. SystemVerilog for Verification. Norwell, MA: Springer, Inc, 2006.

[5] Bromley, Jonathan. "Towards a Practical Design Methodology with SystemVerilog Interfaces and Modports," Conference on Using Hardware Design and Verification Languages, 22 Feb. 2007. San Jose, CA, Accellera 2007

[6] Fitzpatrick, T, D. Rich, and A. Rose. AdvancedVerification Methodology. Ed. Mark Glasser. 3rd ed. Willsonville, OR: Mentor Graphics, 2007.

[7] Bergeron, Janick, E. Cerny, A. Hunter, and A Nightingale. Verification Methodology Manual for SystemVerilog. Norwell, MA: Springer, Inc, 2005.

[8] Cummings, Cliff, and A. Salz. SystemVerilog Event Regions, Race Avoidance & Guidelines. Synopsys User Group, 21 Sept. 2006, Synopsys. 5 Dec. 2007 http://www.sunburst-design.com/papers/CummingsSNUG2006Boston_SystemVerilog_Events.pdf

[9] "URM Class-Based SystemVerilog Library Reference", Cadence Design Systems Inc, San Jose, CA, October 2007

[10] AMBA3 APB Protocol v1.0 Specification, ARM Limited, Cambridge, England, August 2004. ARM Document number IHI0024B

[11] Bromley, J. "Synchronizing a BFM with its clock, without wasting a clock cycle". Available at http://www.svug.org/TipsTricks/tabid/82/Default.aspx

[12] "Questa 6.3c" tool from Mentor Graphics Inc, Beaverton, OR

[13] E. Gamma, R.Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Reading Massachusetts, 1995.