A bottom-up approach to top-down VMM

Xavier Caron

Atmel (Rousset, France)

<xavier.caron@rfo.atmel.com>

ABSTRACT

The building from scratch of a SystemVerilog testbench using VMM methodology.

Stressed DUT is the AHB-Lite compliant bus matrix Atmel IP which boasts crossbar and arbitration features. A bottom-up approach was used starting from DUT through interface, transactor, channel, generator, monitor, and scoreboard objects. Since SystemVerilog+OpenVera interoperability is not yet available – but scheduled for VCS 2006.06 release – AMBA VIP instances had to be replaced with home-made AHB master, slave and monitor alter ego. These makeshift implementations are most likely to be eventually replaced by their VIP counterparts. Coverage techniques are also used to get a qualitative and quantitative feedback on actually simulated transactions, hence on each corner-case sets of constraints.

Besides pipe-cleaning, our primary aim is to learn through this ongoing experiment SystemVerilog, AMBA VIP and VMM. This will in turn help us to assert VMM acceptance within Atmel design or protoverification groups.

Table of Contents

1 Introduction....3
2 Overview....4
2.1 Stressed IP....4
2.2 Testbench at a glance....4
2.3 First chunk of SystemVerilog code....5
3 Testbench architecture....6
3.1 Interfaces (part I)....6
3.2 Test program....6
3.3 Environment....7
Data....8
Behavior....9
3.4 Interfaces (part II)....10
Virtual interfaces....10
Modports and clocking blocks....10
Interfaces as methods holders?....11
3.5 Transactors....11
Mediation-oriented devices....11
The master transactor....12
DIY atomic monitor limitations....13
DW VIP issues....13
3.6 Transactions....14
Objects....14
Scenarii....15
Constraint solving tricks....17
Scoreboard....18
3.7 Functional coverage....19
Coverpoints....20
Cross-coverage....21
Closed coverage events DB....21
4 Yet under construction....22
4.1 As per VCS 2005.06-SP1 release....22
4.2 As per VCS 2006.06-B release....22
5 Conclusions & acknowledgments....23
5.1 Conclusions....23
Overall....23
VMM....23
SystemVerilog....23
AMBA VIP....23
5.2 Acknowledgments....23
6 Appendix....24
6.1 References....24
6.2 Acronyms & links....24
6.3 Verilog DW VIP workaround....25

Table of Figures

Figure 1Coverage-driven CRT paradigm loop....3
Figure 2HMATRIX1 IP....4
Figure 3DUT, master/slave interfaces and test program....4
Figure 4UML class diagram of “env” class (AHB only)....8
Figure 5Environment objects & media....9
Figure 6DW VIP usage (monitor+1 bus vs monitors+n busses)....14
Figure 7Object factory....16
Figure 8Master aggregated vs slave atomic transactions....19

1 Introduction

IP verification is casually addressed by RTL designers with "conventional" techniques based on "good old" Verilog directed testing. Problem is, this kind of testing only copes with what the simulator is explicitly told to check. A way to address this inherent limitation is by using coverage-driven CRT ("Constrained Random Testing") techniques. This way, the testbench randomly "pushes" the DUT ("Device Under Test") in every possible direction, the only limitations being simulation time and constraints sets. Emerging technologies like SystemVerilog – or not-so emerging ones like Vera or E – allow for such a verification paradigm shift. This emergence comes with its own bundle of technical incertitudes but also questions (again) who is actually doing the verification job. Is this still part of an RTL designer prerogatives or should it be delegated to a dedicated verification engineer?

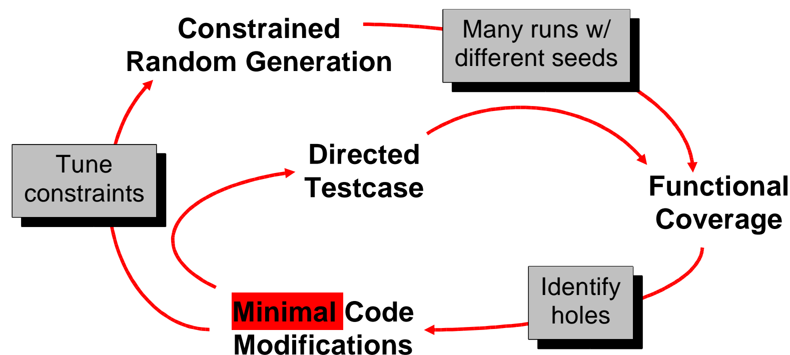

Figure 1Coverage-driven CRT paradigm loop

We already had a promising experience with Vera+RVM+VIP combo. Vera gets the job done, RVM ("Reuse Verification Methodology") is – albeit complex – cleverly designed and used VIPs ("Verification IPs") were stable enough for our requirements. But there is a downside since both RVM and its Vera language implementation are proprietary. Obviously, we would rather use a *potentially* vendor-free and industry-standard one. *Potentially* is the key word as we'll see below...

SystemVerilog language – as per IEEE P1800 LRM ("Language Reference Manual") – seemed to be indeed what we needed. We wanted to keep on using sensible RVM – under its VMM ("Verification Methodology Manual") evolution – and field-proven VIPs. So we devised another experiment, a kind of "proof of concept" about using SystemVerilog testbench-related features, hot off-the-press VMM and VIPs. We "wanted to believe" SystemVerilog to be our next language of choice for verification purposes although its CAD tool support is not total yet. To be fair, as we were writing this paper, no other CAD tool than VCS simulator supported the – mostly class-oriented – SystemVerilog subset required by VMM.

VMM is a meaty methodology indeed which skyrockets quite high in terms of abstraction. The purpose of this paper is to decipher it by starting from the bottom DUT to progressively climb up the abstraction ladder.

2 Overview

2.1 Stressed IP

Atmel proprietary HMATRIX1 IP was used as a pipe-cleaner. It is an APB-configurable, AHB-Lite compliant bus matrix with crossbar and arbitration features. It was chosen mainly for its mid-level complexity which allowed us to concentrate more on testbench-related features than sheer IP-level tricks.

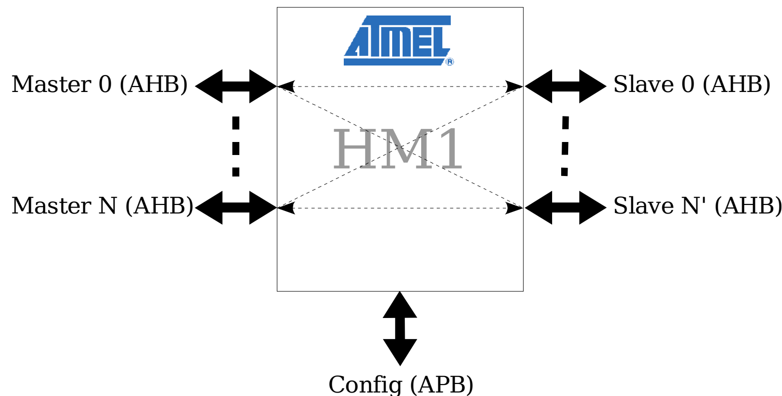

Figure 2HMATRIX1 IP

This rather protocol-oriented IP had one additional challenge – namely its genericity – because of it's RTL source code, automatically generated by a Perl script. As we'll see later on, this would impact chosen TB ("testbench") structure.

2.2 Testbench at a glance

A SystemVerilog, VMM-compliant TB is most likely to be complex. Like in any kind of engineering activity, complexity is to be wrangled by using abstraction and well-defined interfaces. The former is achieved through OO ("Object Oriented") paradigm and so is the latter with additional SV-flavored sugar like interfaces and API ("Application Procedural Interface") scheme extensions like callbacks.

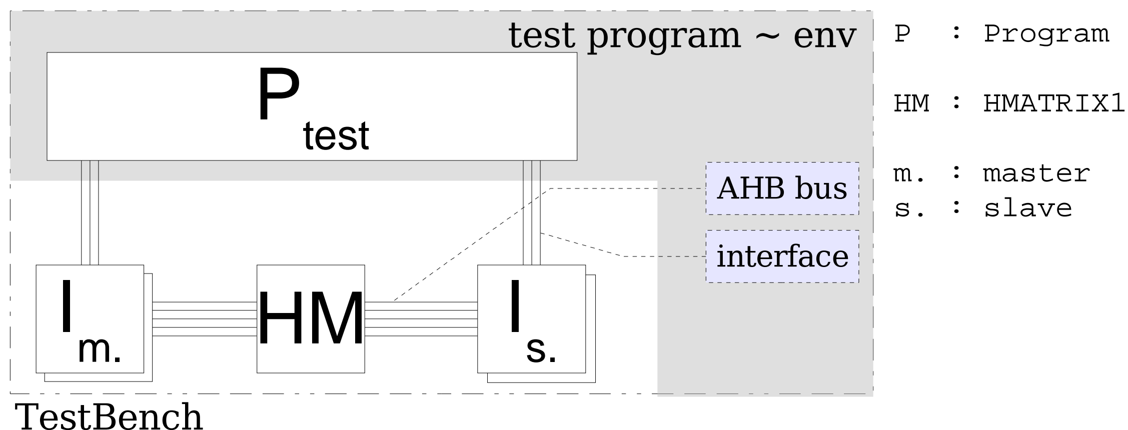

Figure 3DUT, master/slave interfaces and test program

Main “top” object is the automatically generated TB *module*. It instances generic “objects” like *interfaces*, DUT *module* and test *program*. Only the latter is by design subject to being replaced at compile-time to implement different set of constraints, hence different scenarii.

2.3 First chunk of SystemVerilog code

A SystemVerilog code representation of our testbench structure[1] is like, trimmed-down for clarity's sake:

```
module tb_top;
  parameter simulation_cycle = 10;

  bit SystemClock;

  // Program.
  test test_program ( apbi, ahbmvis, ahbsvis );

  // Interfaces.
  hmatrix1_apb_i   apbi          ( SystemClock ); // APB config
  hmatrix1_ahbm_vi ahbmvis [0:1] ( SystemClock ); // AHB master (virtual)
  hmatrix1_ahbs_vi ahbsvis [0:1] ( SystemClock ); // AHP slave  (virtual)

  // DUT.
  hmatrix1 dut (
    // ...
  );

  // Clock.
  always #(simulation_cycle/2) SystemClock = ~ SystemClock;
endmodule
```

Unsurprisingly, a SystemVerilog testbench is not so different from its Verilog parent. It instanciates likewise the DUT and generates the system clock the “old way” but also makes use of pure SystemVerilog constructs like program and interfaces. We can see from this first code excerpt that a port-less[2] VMM SV TB has the following roles:

- define system clock[3]
- instance ad'hoc interfaces
- instance DUT and – if applicable – connect its signals to interfaces
- instance test program and connect it to interfaces

We will take a closer look at all these constructs in subsequent sections.

1 VMM “TESTBENCH INFRASTRUCTURE” chapter, p 103

2 VMM Rule 4-13, p 112

3 VMM Rule 4-15, p 114

3 Testbench architecture

3.1 Interfaces (part I)

Interfaces are low-level instances which reside between DUT module and test program. At first sight, we'll need at least master, slave and monitor kind of interfaces for which driving & sampling mechanism as well as sheer pin directions are most likely to be different. As per VMM paradigm, test program instances transactors, in turn connected to interfaces. We chose to use slave and monitor interfaces – i.e., instead of a direct connection – for coherency's sake and because we needed to glue Verilog '95 flavored IPs.

Our IP RTL source actually does not use interfaces at all, mainly because of “sub-optimal” support by some synthesis CAD tools. TB module is consequently also in charge of gluing IP Verilog '95 pin-oriented connection paradigm with SystemVerilog interface-oriented one:

```
module tb_top;
  // ...

  // Interfaces.
  hmatrix1_apb_i   apbi          ( SystemClock ); // APB config
  hmatrix1_ahbm_vi ahbmvis [0:1] ( SystemClock ); // AHB master (virtual)
  hmatrix1_ahbs_vi ahbsvis [0:1] ( SystemClock ); // AHB slave  (virtual)

  // DUT.
  hmatrix1 dut (
    .hclock      ( SystemClock       ),

    .haddr_ml0   ( ahbmvis[0].haddr  ),
    .haddr_ml1   ( ahbmvis[1].haddr  ),
    .haddr_ram0  ( ahbsvis[0].haddr  ),
    .haddr_ram1  ( ahbsvis[1].haddr  ),
    .hburst_ml0  ( ahbmvis[0].hburst ),
    .hburst_ml1  ( ahbmvis[1].hburst ),
    .hburst_ram0 ( ahbsvis[0].hburst ),
    .hburst_ram1 ( ahbsvis[1].hburst ),
    // well, you get the picture...
  );

  // ...
endmodule
```

Anyway, It seems like we have to make a detour to understand a bit more the test program in order to properly define our interfaces.

3.2 Test program

The test program defines configuration, constraints on transaction object and callbacks. Code wise, it is like:

```
program test (
  hmatrix1_apb_i   apbi,
  hmatrix1_ahbm_vi ahbmvis [0:1],
  hmatrix1_ahbs_vi ahbsvis [0:1]
);
  // Definition of subclasses and callbacks.
  class env_hmatrix1_config extends hmatrix1_config;
    // ...
  endclass

  class env_ahb_data extends ahb_data;
    // ...
  endclass

  class env_ahb_master_xactor_callbacks extends ahb_master_xactor_callbacks;
    // ..
  endclass

  // Core.
  initial begin
    static env the_env = new ( apbi, ahbmvis, ahbsvis, "00-basic" );

    // Configure...

    the_env.run ();

    $finish;
  end
endprogram
```

We're still not too high, abstraction speaking. We've not even encountered the dreaded “vmm_” class prefix but this is not going to last. As we can clearly see, a test program is all about kicking off an “environment” instance[4], provided the objects it operates upon have been properly defined class-wise, since all customization is done in VMM through sub-class and callback concepts. Agreed, we have cheated a bit knowing in advance our `env` instance actually operates – this could count as yet another API OO concept extension – on `env_hmatrix1_config`, `env_ahb_data` and `env_ahb_master_xactor_callbacks` classes.

Bottom line is, to further understand our testbench we need to poke further into this `env` beast in order to know how it operates, on what kind of objects and how it could be customized to define scenarii.

3.3 Environment

The environment object is derived class-wise from its `vmm_env` father. It is roughly a big compound sequencer operating on inter-related object instances. To be true to OO analysis, this class needs a twofold definition, namely one for data and another one for behavior.

4 VMM “SIMULATION CONTROL” section, p 124

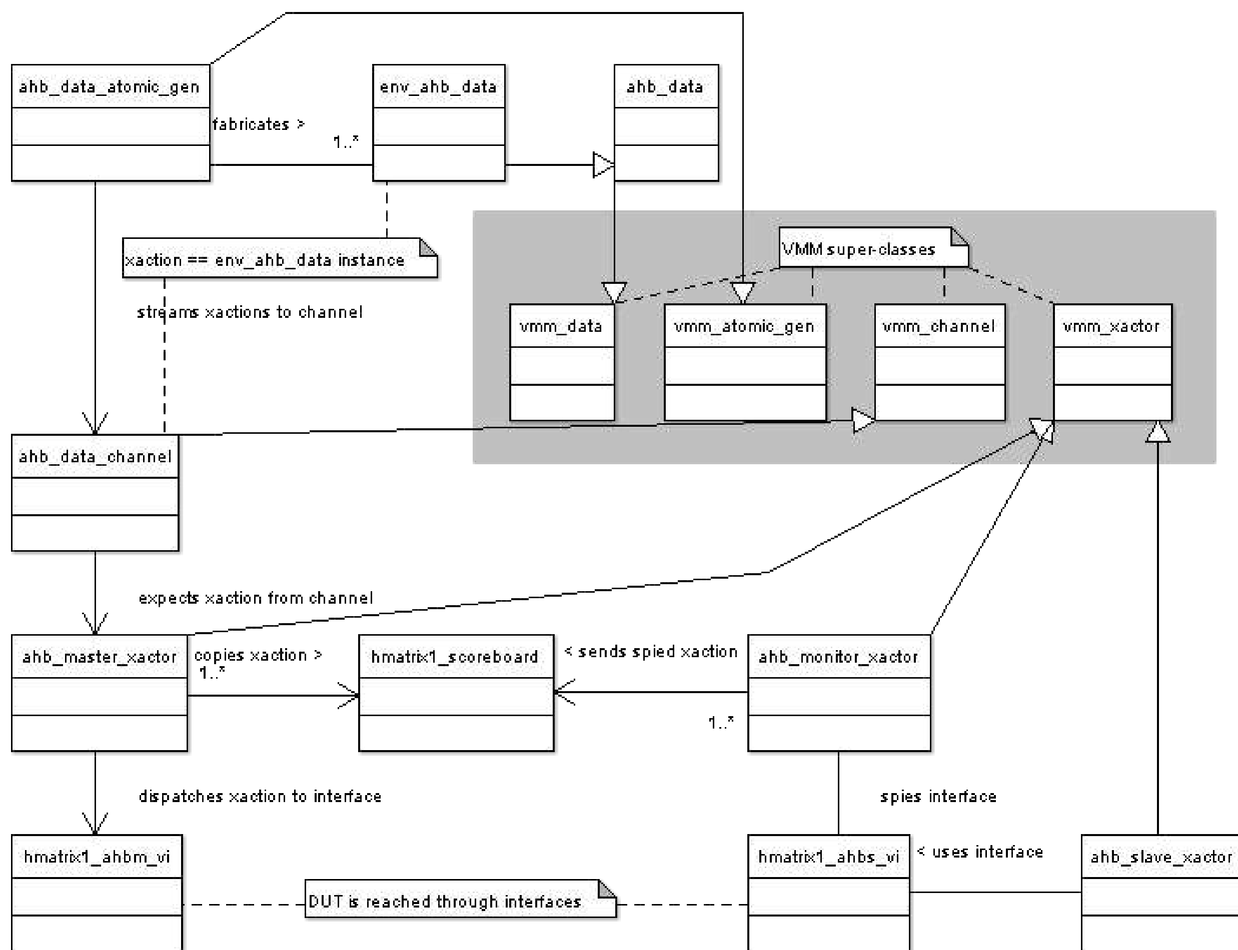

Figure 4UML class diagram of "env" class (AHB only)

Data

The environment object instance is defined by its properties' values. Most of these properties are "handles" to potentially arrayed object instances. Actual definition of testbench structure is done by defining which instance – of what class – talks to which other instance(s).

Transactions are object instances from an atomic *generator* – i.e., an object factory – fed to a master *transactor* through a dedicated point-to-point communication medium[5], a *channel*. The master systematically copies received transactions to the *scoreboard* through callbacks. An alternative design would be to put an extra *monitor* set on masters' side interface to be free from this carbon-copy operation and allow for replacement of "fake" transactor by a "real" implementation IP[6]. I believe we will probably switch to this approach when VIP issues are solved as we will see later on.

We chose to split monitor functionality from slave one. This would likewise allow later on the replacement of a mock transactor by a real instance. We did not do the same split on masters' side mainly because of transactions. Master transactors receive aggregated transactions from the generator. We wanted to instance VIP monitors on slaves' side which feature the ability to report

5 VMM "TRANSACTION-LEVEL INTERFACES" section, p 171

6 VMM Suggestion 5-18, p 226

aggregated transactions. Eventually, scoreboarding would be a mere one-to-one cross-check between masters' and slaves' recorded transactions.

The following figure defines the masters' side instance chains (from generator to transactor), the slaves' side (transactor and monitor in parallel) and the scoreboard as the focal point:

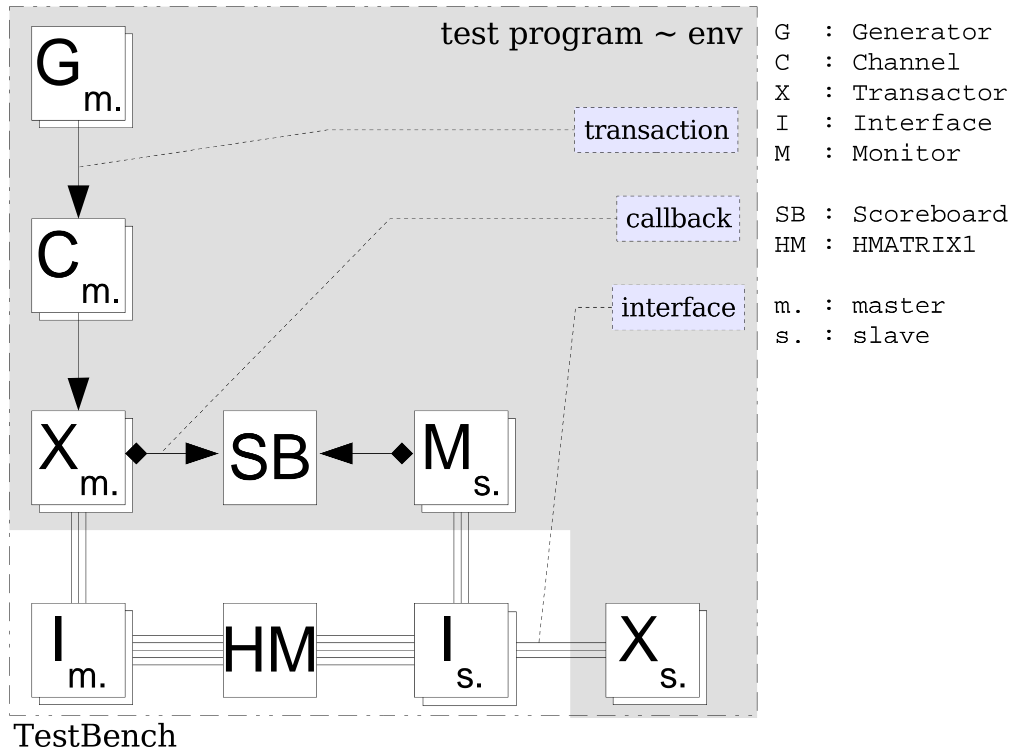

Figure 5Environment objects & media

The generator is an object factory which generates streams – i.e., sequences – of randomized objects. This is both a simple and powerful approach as we'll see later on a section dedicated to transactions.

Behavior

The environment is triggered from the test program by a simple `run` task invocation. This task iteratively sequences through all test stages[7]. Each stage is defined by DWIM-named methods like `gen_cfg`, `build`, `reset_dut`, `cfg_dut`, `start`, `wait_for_end`, `stop`, `cleanup` and `report`. As per VMM book[8], defining our custom environment is all about subclassing all – or part of – these methods as suggested in the following boilerplate code:

7 VMM “SIMULATION CONTROL” section, p 124

8 VMM Rules 4-30 & 4-31, p 128

```
class env extends vmm_env;
  // Properties.
  // ...

  // Bootstrap.
  task run ();
    super.run ();
  endtask

  // Stages.
  function void gen_cfg ();
    // Call superclass counterpart.
    super.gen_cfg ();

    // Customization starts here!
    // ...
  endfunction

  // ...
endclass
```

3.4 Interfaces (part II)

Virtual interfaces

A bus matrix has – by design – a replicated interface schema. We wanted to respect this particularity by using *virtual* interfaces[9] and thus be able to connect each AHB transactor to its virtual interface counterpart. We therefore instanced an *array* of virtual interfaces talking to an *array* of transactor objects. Each transactor was in turn connected to its matching interface through ad'hoc constructor call:

```
for ( int i = 0 ; i < masters.size ; i++ ) begin
  ahb_master_channels[i] = new ( "ahb_master_channel", ... );
  ahb_master_xactors[i]  = new ( ahbmvis[i], ahb_master_channels[i], ... );
end
```

We initially wanted to have an associative array of transactor instances because each master or slave is actually named at chip and RTL configuration levels. Indeed, “foo["arm0"]” would have been clearer than “foo[0]” (i.e., a mental indirection level less, it never hurts sparing braincells a bit anyway) but SystemVerilog specification only supports *arrayed* (with integer indices) interfaces instances and not *hashed* (with string indices) ones. We nevertheless kept this indirection by including the following automatically generated code snippet:

```
string masters[$] = { "ml0", "ml1" };
string slaves[$]  = { "ram0", "ram1" };

string ahbmvii[*];
string ahbsvii[*];

`define populate_ahbmvii_hash ahbmvii["ml0"] = 0; ahbmvii["ml1"] = 1;
`define populate_ahbsvii_hash ahbsvii["ram0"] = 0; ahbsvii["ram1"] = 1;
```

Modports and clocking blocks

Hot topics about interfaces are modports[10] and clocking blocks[11]. They were used to further constrain the rather lax all-wire interface definition scheme which eventually leads to the

9 VMM Rule 4-108, p 169

10 VMM Rules 4-9 & 4-10, p 110-111

11 VMM Rules 4-10, 4-11 & 4-12, p 111-112

dreaded, unrestricted all-inout connection scheme (i.e., check out the "`Notice: Ports coerced to inout, use -notice for details`" message from VCS). Three kinds of modports / clocking blocks combo were actually needed and used:

- TB-like (reversed DUT direction-wise) and clocked (except asynchronous reset pin).

  Used with *master* transactors for TB driving and sampling purposes.
- DUT-like and unclocked.

  Used with *slave* transactors.
- All-in and unclocked.

  Used with *monitor* transactors.

Modports and clocking blocks were invaluable as sanity checks per se and because it forced us to precisely define our TB architecture by answering the rather straightforward question: who talks to whom and through which medium?

Interfaces as methods holders?

Interfaces – as per SystemVerilog LRM – allow for function and task definition. As we'll see later on in this paper, makeshift transactors had to be coded in order to cope with lack of SV+OV interoperability. What had been coded at BFM ("Bus Functional Model") class-level (i.e., a master transactor wiggling DUT pins through an interface) could have been abstracted further by delegating such low-level operations to interface tasks or function calls. Anyway, one aim of this whole pipe cleaning was to eventually be able to use "out of the box" VIP objects and not losing time devising tedious pieces of code like AHB monitors. This is definitely no trivial task and we are not a software company anyway...

3.5 Transactors

Mediation-oriented devices

Transactors[12] are mediators between an abstract representation of a transaction (i.e., an object) and its physical expression through temporal, protocol-compliant pin wiggling. We coped with three kinds of transactors, namely master, slave and monitor. A master transactor for example usually receives object instances – again, transactions are objects – sent via a channel (i.e., a queue) and "transcode" its content into timed waveforms via an interface (modport or clocking block pin wiggling or yet another layer of abstraction through method calls). In other terms, they react to transactions and act through interfaces. Note a monitor transactor has exactly the opposite role. It spies wiggling pins, tries to decipher beats and eventually aggregates bursts to be sent – usually to the scoreboard – either via a channel or through a callback[13].

Transactors are a key concept in VMM abstraction scheme. They are usually meant – at least in Synopsys world – to be VIP instances. Problem is, SystemVerilog – under its VCS 2005.06-SP1 implementation – did not talk very well with OpenVera modeled objects like VIP. We consequently had to code our own master, slave and monitor makeshift transactors. Writing AHB-Lite compliant masters and slaves was not too hard, although we had to code features like controlled ability to respectively deliver busy or unready states. The real challenge was to write the monitor transactor. Coding a monitor is a bit akin to writing a parser. Both operate on a stream and try to recognize sequences in order to build higher-level constructs, here an object

12 VMM "TRANSACTORS" section, p 161

13 VMM Recommendation 5-48, p 249

instance. But the monitor – acting on a real time stream – cannot possibly use conventional parser techniques such as back-tracking or lookahead. Since our transactors were definitely intended to be transient, we decided to simplify the code to be written by reporting atomic-level transactions only. E.g., one INCR8 master transaction would expect 8 SINGLE counterparts, actually from an unbroken INCR8 on slave's side or maybe two consecutive INCR4 for example. Monitor-wise, this would not matter. The scoreboard would then eventually be in charge to establish the link between both sides: one to many relationship, (un)broken stream, etc.

The master transactor

A master transactor is essentially a big dispatch machine waiting forever. It is generic because each transactor instance waits on its channel[14] and reacts accordingly on its virtual interface. Both are properties defined by constructor's call: channel is connected to generator, interface to DUT. Another way to improve the transactor's genericity is by embedding tasks calls (aka "callbacks") at key points[15] within `main` execution loop. Callbacks are then to be registered during `env::build` stage by calling superclass' `append_callback` method.

```
task ahb_master_xactor::main ();                           // AHB_MASTER CLASS
  super.main ();

  forever begin
    ahb_data xaction;

    `vmm_callback ( ahb_master_xactor_callbacks,       // PRE-PEEK CB
      pre_peek_callback ( this ) );

    i_channel.peek ( xaction );

    `vmm_callback ( ahb_master_xactor_callbacks,       // PRE-DISPATCH CB
      pre_dispatch_callback ( this, xaction ) );

    case ( xaction.xact_type )
      ahb_data::IDLE  : idle;
      ahb_data::WRITE : wiggle_pins ( xaction, 1 );
      ahb_data::READ  : wiggle_pins ( xaction, 0 );
    endcase

    `vmm_callback ( ahb_master_xactor_callbacks,       // PRE-GET CB
      pre_get_callback ( this ) );

    i_channel.get ( xaction );
  end
endtask

// Meanwhile, in another file...

function void env::build ();                               // ENV CLASS
  super.build ();

  // ...

  for ( int i = 0 ; i < masters.size ; i++ ) begin
    env_ahb_master_xactor_callbacks callback = new ( scoreboard );

    ahb_master_xactors[i].append_callback ( callback );
  end
endfunction
```

14 VMM Rule 4-111, p 172

15 VMM Rules 4-154 & 4-159 + Recommandations 4-155, 4-156, 4-157 & 4-158, p 198-199

This is actually an extension of the concept of API – albeit an usual one in the software world – for genericity's sake. I mean, in OO paradigm, an object encapsulates private data and provides public methods. Hence one key to use an object is knowing its set of properties (ID, type, public or private, etc.) and associated methods (constructor, accessors, mutators, etc.) In VMM methodology we also need to know when (i.e., at which code stage or milestone) and with which prototype callback functions (i.e., `` `vmm_callback `` macro call) could be overridden. Callbacks are virtual methods pertaining to a subclass of `vmm_xactor_callbacks` class[16]. Since – and unlike some other languages – SystemVerilog functions are not treated as first-class citizens, using this `` `vmm_callback `` trick[17] – with `this` handle to calling transactor[18] – is the only generic way to inject code into a method without having to subclass (again) `vmm_xactor` subclass' virtual functions.

DIY atomic monitor limitations

As seen above, we started with the intention of coding a rather lightweight monitor transactor in the true DIY ("Do It Yourself") tradition. One way of keeping it lightweight was to report atomic (i.e., beat-wise) transactions only. Problem is, this works well enough as long as you do not care about data integrity or slaves' side broken streams.

Data integrity is about checking if data to-be-written fed on masters' side eventually reached slaves' side, or vice-versa in the case of a read operation. As per AMBA protocol, a data phase begins after an address phase has been seen. For example, data is shifted by one cycle with reference to control when `HREADY` is stuck to high. This means last datum is to be fetched from subsequent transaction result in an atomic scheme. Although post-processing of simulation stored results could indeed aggregate full-blown transactions from atomic ones (refer to "Master aggregated vs slave atomic transactions" figure on page 19), we decided not to waste time coding such a workaround.

Slaves' side broken streams are another issue to cope with. Although, atomically speaking, each beat of a burst (expanded on masters' side) can easily be correlated with its atomic slaves' side beat counterpart, knowing for example that an INCR16 (master-wise) was broken into two INCR8 (slave-wise) indeed matters because it reflects key configurable arbitration features of our bus matrix like not allowing any master to hold the bus for too long.

Bottom line is, what we craved for was VIP availability. We eventually got stuck because of our own makeshift transactors limitations. We then decided to work these around using Verilog VIP instances (i.e., Vera bootstraps) to – at last – have real transactions reported. But this tweaking had a price, namely an extra Vera dependency (thus simulation resources overhead) and a significant decrease in our abstraction level (arrayed object instances were replaced by single module instances + initial blocks + ugly `` `define `` macros). Refer to "Verilog DW VIP workaround" appendix on page 25 for a code chunk sample of such a workaround implementation.

DW VIP issues

At the time of writing this paper, we also had problems with DW VIP monitor about slave bus events happening without any `HSEL` asserted from the matrix. It seems like this is a side-effect of a VIP feature by which the monitor only cares to read `HSEL` value(s) prior to its `start` call. Sadly, this conflicts with one (too?) clever re-arbitration saving feature of our bus matrix.

16 VMM Rule 4-159, p 199

17 VMM Rule 4-163, p 200

18 VMM Rule 4-162, p 200

Indeed, DW VIP monitor is by design meant to be connected to a single bus, in turn potentially connected to more than one slave (hence the *static* HSEL sampling scheme). But for system-on-chip architectures mainly based on a bus matrix backbone – read: all our designs – we would have liked to connect one monitor per slave bus and have a *dynamic* HSEL sampling scheme. This blocking issue has been duly reported to Synopsys R&D. From a conference call with R&D persons, it appeared that DW VIP was not designed as a point-to-point monitoring device but rather as a bus – in the AMBA sense of the word – oriented one. Indeed, the former may be left unselected while the former should not. One suggested workaround – currently under investigation – is about gating HTRANS with HSEL on slaves' side.

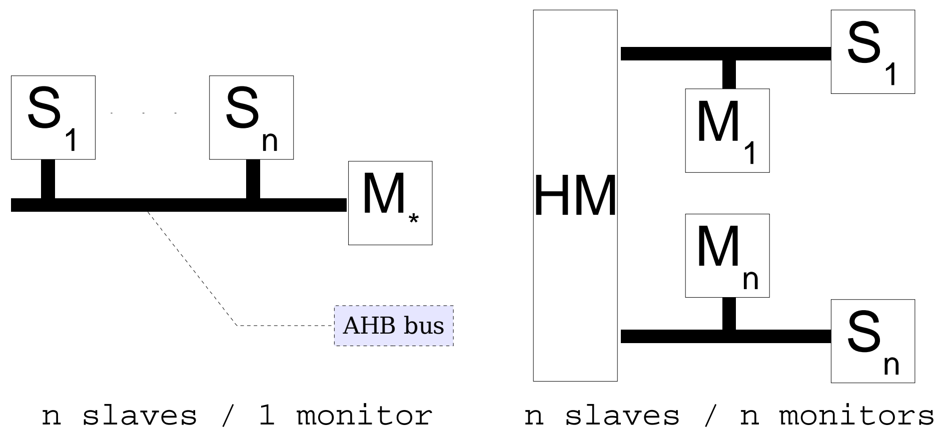

Figure 6DW VIP usage (monitor+1 bus vs monitors+n busses)

Last, as yet another last resort alternative, we replaced our slave SV transactor with its VIP implementation (same workaround trick as above) but DW VIP slave only features beat-level watchpoints. Shifted data is nevertheless handled correctly but we faced other problems trying to have actual HTRANS values for broken streams analysis' sake.

3.6 Transactions

Objects

A transaction is an object instance in VMM paradigm[19] whereas other transaction-level methodologies would rather define it as a mere method call[20]. Being by essence more abstract but also more memory consuming, objects can be streamed over channels (e.g., from generator to transactor) or stored within any container-like object (e.g., a queue in a scoreboard). Objects fundamentally define a namespace with associated methods and properties.

Methods are usually constructors, accessors and mutators. They are also at the very least facilities like display, copy or compare routines (this is demanded and detailed in the VMM book[21]) but can encompass other service functions like the reckoning of addresses based upon base address and burst type for example.

19 VMM Rule 4-53, p 140

20 Contradicts VMM Rule 4-54, p 141

21 VMM Rules 4-76 & 4-77, p 155

Properties define an instance, i.e., in OO terminology VMM objects are said to be "mutable". Although OO speaking instance properties define a "state", we can functionally discriminate these attributes in three groups, depending on their role:

- carry the "payload"
- control transactor behavior
- ease up scoreboarding

The payload part is quite straightforward. For example, to model an AMBA transaction, some involved signals protocol-wise (like `HBURST`, `HSIZE`, etc.) would have to be stored, the object acting like a mere envelope, in order for the end-of-chain BFM to temporally "unroll" these – generating on-the-fly other control signals like `HSEL` if needed – to carry out physical pin activity.

Other properties were on the other hand dedicated to transactor behavior control, for example to randomly force a master into a locked access or to make it response with busy wait states. Properties such as `locked` and `busied` also ease-up scoreboarding since they allow for cross-coverage with payload-oriented properties.

Scoreboarding also requires a set of dedicated properties. For example, `sender_n` and `receiver_n` are used to respectively store master sender number (remember, master transactors are arrayed instances) and intended slave number. Both integers are reckoned by – configuration-dependent – address-space indirection. Only `sender_n` is used on slaves' side. Both properties are heavily used to perform master-to-slave correlations at scoreboard-level but are also invaluable for subsequent cross-coverage operations.

Our `ahb_data` class is an extension of `vmm_data` class. Property-wise, it is reduced to:

```
// Payload:
rand hxact_t      xact_type;  // READ, WRITE or IDLE
rand haddr_t      addr;
rand hdata_t      data [$];   // queued to cope with burst type
rand hburst_t     burst_type; // SINGLE, INCR4, WRAP4, etc.
rand hsize_t      xfer_size;  // BYTE, HALFWORD or WORD
rand hprot_t      protection; // 0x????

// Scoreboarding:
     hdata_t      atomic_data       = 0;
     hburst_t     atomic_burst_type = ahb_data::SINGLE;

     int unsigned sender_n;
     int          receiver_n = -1;
     time         birth_time;

// Transactor control (and a bit of scoreboarding too):
     bit          locked    = 0;
     bit          proted    = 0;
     bit          busied    = 0;
```

Scenarii

From the TB point of view, a simulation run implements one or more scenario, hopefully related to a test plan item. As previously stated, each scenario is a stream of constrained random instances, hence the CRT qualifier for this whole testing paradigm. Depending on which kind of VMM scenario generator base object is chosen (read: "subclassed"), TB might enforce one

scenario[22] (i.e., `vmm_atomic_gen`) or a set of scenarii[23] (i.e., `vmm_scenario_gen`). We have only used the former kind of generator so far[24].

Scenario generators are often referred to as "object factories". They are indeed akin to factories, restlessly replicating constrained random variations of template objects[25]. Each instance of the whole randomized stream is actually an instance of the same base class and thus has its relevant properties randomized. Atomically speaking, each instance pertains to a same "possibilities space" delimited by its base class constraints, in turn a subset of AMBA legal possibilities, in turn a subset of all potential combinations of decorrelated properties:

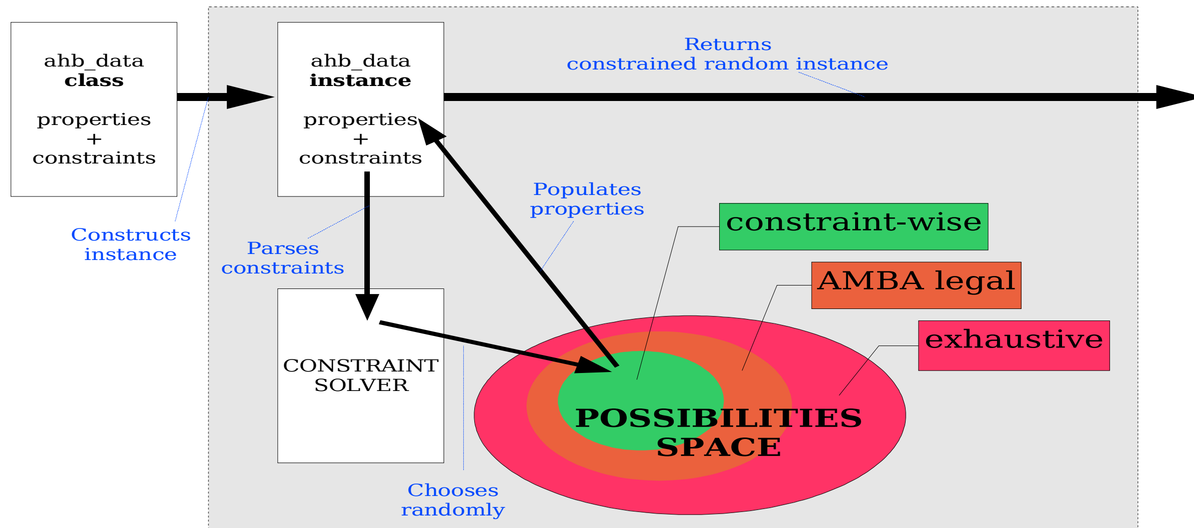

Figure 7Object factory

Our environment works on a subclass of `ahb_data`, namely `env_ahb_data`. The former is generic across corner-cases while the latter is meant to be tweaked on a corner-case by corner-case basis. For example, on one hand our `env_ahb_data` has the following corner-case driven set of constraints:

22 VMM Recommendation 5-23, p 231

23 VMM Recommendation 5-24, p 234

24 VMM "Which Type of Generator to Use?" section, p 244

25 VMM Rule 5-6, p 216

```
// Configuration dependant.
static constraint addresses_per_slave {
  addr inside {
    ['h00000000:'h000fffff],
    ['h00100000:'h001fffff],
    ['h00200000:'h002fffff]
  };
}

// For debugging purposes.
static constraint tiny_data {
  foreach ( data[i] ) data[i] < 1024;
}
static constraint no_wrap {
  burst_type inside { SINGLE, INCR4, INCR8 };
}
static constraint words_only {
  xfer_size inside { WORD };
}
```

While on the other hand our `ahb_data` implements AMBA-driven constraints or TB-related restrictions[26]:

```
constraint word_boundary_multiple_of_sixteen {
  addr [ 2: 0] == 3'b0;
  addr [31:30] == 2'b0;
}

constraint less_than_words_or_words_only {
  xfer_size inside { BYTE, HALFWORD, WORD };
}

constraint the_number_of_the_beats { // FIXME no INCR support
  burst_type != INCR;
  burst_type != INCR -> data.size inside { 1, 4, 8, 16 };
}

constraint data_size_from_burst_type {
  burst_type inside { SINGLE         } -> data.size ==  1;
  burst_type inside { INCR4,  WRAP4  } -> data.size ==  4;
  burst_type inside { INCR8,  WRAP8  } -> data.size ==  8;
  burst_type inside { INCR16, WRAP16 } -> data.size == 16;
}

constraint bursts_cannot_pass_1K_boundary { // WORD only
  data.size ==  1 -> addr[9:0] < 1020;
  data.size ==  4 -> addr[9:0] < 1008;
  data.size ==  8 -> addr[9:0] <  992;
  data.size == 16 -> addr[9:0] <  960;
}
```

Testbench-related restrictions are actually unsupported or yet unimplemented features.

Constraint solving tricks

We encountered constraint solver issues with data queue size versus burst type. The latter actually defines the size of the former. Burst type has to be randomized, then data queue size and last data queue content. This forced us – soundly advised by our Synopsys support person – to use cryptic "`+ntb_solver_mode=1`" run-time switch because the constraint solver is by default in a "fast" mode which choked on the following subset of constraints:

26 VMM Rule 4-80, p 157

```
constraint the_number_of_the_beats { // FIXME no INCR support
  burst_type != INCR;
  burst_type != INCR -> data.size inside { 1, 4, 8, 16 };
}

constraint data_size_from_burst_type {
  burst_type inside { SINGLE         } -> data.size ==  1;
  burst_type inside { INCR4,  WRAP4  } -> data.size ==  4;
  burst_type inside { INCR8,  WRAP8  } -> data.size ==  8;
  burst_type inside { INCR16, WRAP16 } -> data.size == 16;
}
```

Another issue was uncovered about burst (starting) address versus burst type. Since bursts lengths cannot possibly exceed a 1K boundary (as per AMBA specification), we had to code the following constraint:

```
constraint bursts_cannot_pass_1K_boundary { // WORD only
  addr[9:0] + ( 4 * data.size ) < 1024;     // FIXME thrashes CS
}
```

Problem is, this makes the constraint solver go wild and completely de-correlate `data.size` versus `burst_type`. We had to (temporarily?) replace it – at the cost of INCR transactions support – by:

```
constraint bursts_cannot_pass_1K_boundary { // WORD only
  data.size ==  1 -> addr[9:0] < 1020;
  data.size ==  4 -> addr[9:0] < 1008;
  data.size ==  8 -> addr[9:0] <  992;
  data.size == 16 -> addr[9:0] <  960;
}
```

Scoreboard

The scoreboard was coded in our experiment as a passive instance collecting events from masters (duplicated from transactors) and slaves (as decoded by monitors) during the whole simulation[27]. No such cheesy thing as a parallel scoreboard reactively driving a feedback loop with generators by tweaking constraints in RT based upon instant coverage results. We did not currently see the need for such a sophisticated technique, but maybe we were just not advanced enough in verification lore...

Our scoreboard performed “a posteriori” cross and sanity checks during the report stage of the run. As we've already seen earlier, one issue to solve was the plain (i.e., aggregated) vs atomic (i.e., beat-wise) nature of reported transactions, depending on which “side” of the matrix triggered the actual recording.

When the simulation is complete, our scoreboard has two queues of transactions to handle. We first intended to use associative arrays but we did not find any relevant way to actually hash transactions. Having potentially any number of concurrent masters issuing random addresses transactions ruled out a simple address-based algorithm. Also, any hashing scheme loses by essence the order in which transactions were actually pushed. This would be a problem when tackling verification of arbitration features for which relative order of transactions – or broken sub-transactions on slaves' side – would be of utmost concern. Downside of this approach is the $O(N^2)$ search time, although it is actually cut down a bit by taking transaction “birth time” into account to ignore a loop item or break out of the loop.

Adoption of VIP monitor could enable us to revise this accumulation based scheme. A dynamic one-to-many transaction-wise comparison could be implemented instead – thanks to aggregated transaction reports – which would in turn allow for matched objects freeing during simulation.

27 VMM “FEEDBACK MECHANISM” section, p 277

3.7 Functional coverage

The scoreboard did a *quantitative* study on our transactions sets. Functional coverage[28] is a means to have a *qualitative* feedback on generated transactions.

We defined a covergroup within our `ahb_data` class. This gave useful results but had the limitation of covering transactions from masters' point of view. For example, there is currently no possible coverage on slave-driven events like unready states which a slave may legally insert within its response stream. Such intermediate states are not currently handled by our monitor implementation and thus do not leave a trace transaction-wise at scoreboard level. This could have been implemented but would have somehow conflicted with the essentially beat-level nature of our scoreboard. Unlike slaves' side transactions, busy master instances are tagged through a dedicated property because such transactions are driven from masters' side. The `busied` boolean-typed property is hence populated *a priori* by a generator and not *a posteriori* by a monitor. From a coverage point of view, we indeed know a master transaction contained busy states but we don't know neither how many were inserted nor when these busy states occurred. The `busied` property is not used at scoreboard level but at coverage level only.

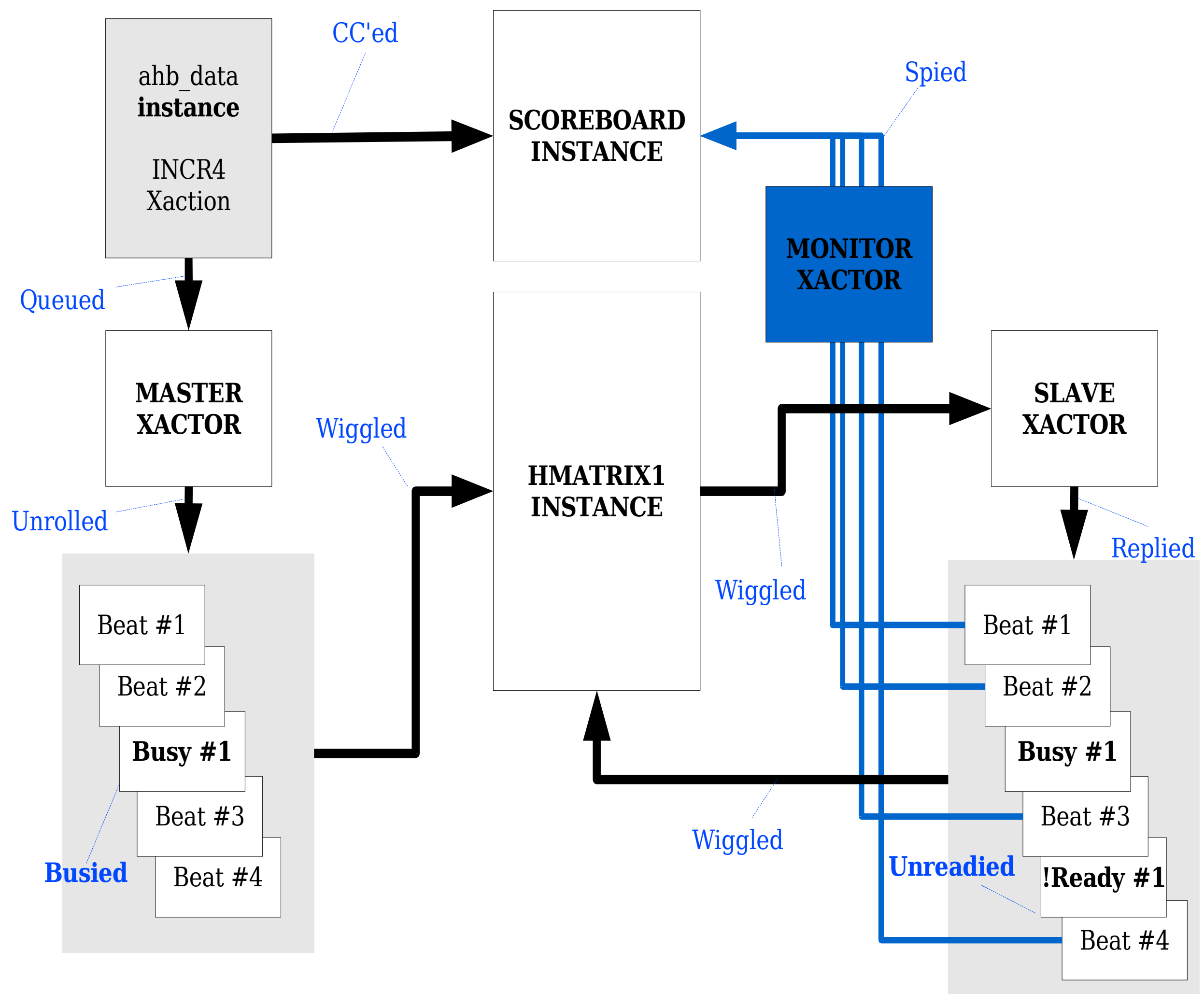

Figure 8Master aggregated vs slave atomic transactions

28 VMM “COVERAGE-DRIVEN VERIFICATION” chapter, p 259

Coverpoints

An excerpt from our `ahb_data` covergroup[29] definition follows:

```
covergroup xactions_cg ();
  xact_types : coverpoint xact_type {
    bins idle = { IDLE  }; bins read  = { READ  }; bins write = { WRITE };
  }
  rw_xactions : coverpoint xact_type {
    bins rw = { READ, WRITE };
  }
  addrs : coverpoint addr {                              // From configuration.
    bins to_slave_0__ml0 = { ['h00000000:'h000fffff] };
    bins to_slave_1__ml1 = { ['h00100000:'h001fffff] };
  }
  burst_types : coverpoint burst_type {
    bins         single                 = { SINGLE };
    bins         incr4                  = { INCR4  };
    bins         wrap4                  = { WRAP4  };
    bins         incr8                  = { INCR8  };
    bins         wrap8                  = { WRAP8  };
    bins         incr16                 = { INCR16 };
    bins         wrap16                 = { WRAP16 };
    illegal_bins not_yet_supported_incr = { INCR   };
  }
  xfer_sizes : coverpoint xfer_size {
    bins         bytes                 = { BYTE     };
    bins         halfwords             = { HALFWORD };
    bins         words                 = { WORD     };
    illegal_bins not_implemented_by_ip = { TWO_WORDS, FOUR_WORDS_LINE,
      EIGHT_WORDS_LINE, SIXTEEN_WORDS_LINE, THIRTYTWO_WORDS_LINE };
  }
  protections : coverpoint protection {
    wildcard bins data_access        = { 'b???1 };
    wildcard bins priviledged_access = { 'b??1? };
    wildcard bins bufferable         = { 'b?1?? };
    wildcard bins cacheable          = { 'b1??? };
  }
  locked : coverpoint locked {
    bins   locked = { 1 };
    bins unlocked = { 0 };
  }
endgroup
```

Coverpoints are rather straightforward to define. Most coverpoints' bins are mere lists of enumerates or integer ranges but some others are defined using the clever `wilcard` scheme. A semantically interesting feature is reached through the `illegal_bins` keyword. We chose to use this keyword – instead of its `ignore_bins` cousin – because we indeed intended to rule-out certain possibilities but with an extra twist[30]. We also wanted to convey the idea of categories that were not to be cared about because of either lack of IP implementation or missing / yet to be coded TB features. In this respect there is a not-so-obvious relationship between the constraints and the covergroups respective lineups. We sometimes faced difficulties trying to interpret coverage results just because our constraints and covergroups definitions were out of phase with one another.

29 VMM Rule 6-7, p 266

30 As per SystemVerilog LRM, populating an `illegal_bins` kind of bin triggers a run-time error.

Cross-coverage

Cross-coverage was invaluable to answer questions like "did I exert all possible burst types in both read and write mode?" or "what proportion of locked transfers did the run perform?" Cross-coverage actually allowed us to close the loop (see "Coverage-driven CRT paradigm loop" illustration on page 3) by incrementally – and manually[31] – tweaking our set of constraints or increasing our simulation runtimes. For example, as per our `ahb_data` class:

```
// Extra coverpoints needed by cross-coverage (from configuration).
coverpoint sender_n {
  bins        master_0__ml0 = { 0 };
  bins        master_1__ml1 = { 1 };
  ignore_bins irrelevant    = { [3:$] }; // sender_n is a natural
}
coverpoint receiver_n {
  bins         slave_0__ram0      = { 0 };
  bins         slave_1__ram1      = { 1 };
  ignore_bins  irrelevant         = { [$:-2], [2:$] };
  illegal_bins to_undefined_slave = { -1 }; // receiver_n is an integer
}

// Cross-covers.
rw_vs_bt           : cross rw_xactions, burst_types;
locked_rw          : cross rw_xactions, locked;
masters_to_slaves  : cross sender_n, receiver_n;
locked_rw_masters  : cross rw_xactions, locked, sender_n;
xfer_size_masters  : cross xfer_sizes, sender_n;
burst_type_masters : cross burst_types, sender_n;
rwi_vs_burst       : cross xact_types, burst_types;
```

Closed coverage events DB

An issue with coverage is about the event proprietary database. Being binary and proprietary forced us to use front-end text or HTML translators bundled with VCS. We would have preferred having a more open kind of database – ideally with a formal API – in order to be able to post-process it from a high-level language like Perl.

31 VMM Recommendation 6-26, p 277

4 Yet under construction

Our experiment is obviously not yet completed. On one hand some items are not yet implemented because we're still learning SV language or AMBA protocol tricks. On the other hand some others are gated by VCS or VIP improvements.

Benchmarking the used resources – memory and runtime – is also still on the "to do" list. In this respect, we want to assert for sheer memory usage with special care on potential memory leaks issues.

4.1 As per VCS 2005.06-SP1 release

The following items are currently still to be coded:

- Handle ERROR response at master and slave levels. We are still not convinced we should code it in our makeshift transactors since VIP slave does not support this feature either.
- Clean APB configuration scheme through a dedicated transactor. This is not as simple as it seems since DUT configuration impacts the scoreboard behavior with reference to high-level arbitration features such as broken streams on slaves' side.
- Datum check (linked to VIP monitor `HSEL` issue). Using VIP monitor should allow a straightforward one-to-one cross-check (including data integrity).
- Support INCR bursts (partially linked to constraint solver issues).

4.2 As per VCS 2006.06-B release

VCS 2006.06-B pre-release was made available to us at the beginning of March. We are quite eager to try our VIP monitor flavored TB but we believe we will still have to cope with the `HSEL` issue. We hope the gating of `HTRANS` by `HSEL` on slaves' sides will be a usable solution for this blocking issue.

We will then have to replace all our transactors by their respective VIP counterparts. Replacing our homemade transactors by VIP ones will dramatically change our testbench on an object-by-object basis but not structurally speaking. It seems like genericity starts to pay off sooner than expected.

First and foremost, our `ahb_data` class – the actual underlying testbench backbone – will have to be changed into a subclass of `dw_vip_ahb_master_transaction` instead of current `vmm_xactor`. As a corrolary, its properties types and identifiers are going to change since DW VIP uses formally[32] constructed identifiers like, for example:

```
DIY        -> VIP
addr       -> m_bvAddress  : bv  ~ bit vector
xact_type  -> m_enXactType : en  ~ enumerated
data       -> m_bvvData    : bvv ~ bit vector vector (array of bv)
```

Likewise, we will have to turn our transactors into `dw_vip_ahb_<whatever>_rvm` subclasses (replace `<whatever>` with `master`, `slave` and `monitor`).

32 Using "Hungarian notation", refer to Wikipedia for more.

5 Conclusions & acknowledgments

5.1 Conclusions

Overall

One goal of this whole experiment was to pipe-clean a new verification paradigm. We eventually wanted to be able to hand out the RTL designer (or the verification engineer) a complete testbench boilerplate generator together with directives about how use-case defining constraints could be "just" tweaked to implement more and more test plan items. Obviously, the experiment is not yet complete but we know the VMM+SV+VIP combo is close to be suitable for our needs. We are confident – once VCS 2006.06 and an improved/worked-around DW VIP monitor are available – about being able to use a very powerful and modular solution indeed.

Yet, one aspect we still have to explore is not a technical but a human one. We are still wondering about the pre-required mindset in order to properly use such a sophisticated software-based approach. On one hand the RTL designer knows (too?) well *his* IP but could balk at very abstract concepts. On the other hand a dedicated verification engineer knows (too?) little about *an* IP but is very comfortable with complex software tricks. We believe we will have to try to hand out the whole testbench structure to an RTL guy when this experiment is stable enough in order to see if he may – or may not – be left to his own devices.

VMM

Despite its steep learning curve, our experiment confirms that VMM is a solid methodology indeed. As an advice, users might want to start with RVM trainings – to be kindly asked to a Synopsys support person – rather than reading the huge VMM book in the first place.

SystemVerilog

VCS support of SystemVerilog was good enough. Yet, we sometimes encountered "`SVNYS: SystemVerilog not yet supported`" error messages but overall, SystemVerilog is a rich language and its VCS implementation was redundant enough to allow for not-so-inelegant workarounds. Alas, we did not have time enough to check these uninplemented features against latest 2006.06-B release.

AMBA VIP

We believe the availability of 2006.06 release with SystemVerilog+OpenVera interoperability is mandatory for our needs. Although – compared to USB for example – AMBA is not a very complicated protocol, coding decent transactors is no trivial task especially when VIPs exist, are field-proven and loaded with features, RVM channel integration among others.

Bottom line is, we needed the VIPs to *really* get the job done.

As a last minute update on this topic, VCS 2006.06-B was made available to us beginning of March. This was a little too late to significantly alter the present paper content but this was nevertheless very good news indeed. We hope we will be able to add a slide or two on this most wanted interoperability feature before the final slide deadline!

5.2 Acknowledgments

A big hand to our local Synopsys support. Thanks to Karim and Fabian for their support. We appreciated your technical skills and short response times!

6 Appendix

6.1 References

The following materials were used during our experiment:

- Verification Methodology Manual for SystemVerilog (aka "VMM book")
- DesignWare AHB Verification IP Databook
- AMBA specification (Rev 2.0) – ARM IHI 0011A
- SystemVerilog 3.1a Language Reference Manual (Accelera's Extensions to Verilog)
- Reference Verification Methodology Tutorial (from VCS tree)
- Reference Verification Methodology User Guide (ditto)

6.2 Acronyms & links

- AHB: "Advanced High-performance Bus" (from AMBA) for high frequency peripherals
- AMBA: "Advanced Microcontroller Bus Architecture" a bus protocol from ARM
- APB: "Advanced Peripheral Bus" (from AMBA) for low-power peripherals
- BFM: "Bus Functional Model"
- DW VIP: "DesignWare Verification IP"
- HDL: "Hardware Design Language"
- HDVL: "Hardware Design and Verification Language"
- LRM: (SystemVerilog) "Language Reference Manual"
- OO: "Object Oriented" one of the 4 main computer science paradigms
- OpenVera, OV: open source hardware verification language, evolution of proprietary Vera
- Perl: "Practical Extraction and Report Language" the swiss-army chainsaw
- RVM: "Reuse Verification Methodology" ancestor of VMM, Vera/OpenVera-based
- SystemVerilog, SV: an HDVL (IEEE P1800 standard), extension of Verilog HDL
- VCS: simulation tool from Synopsys
- VMM: "Verification Methodology Manual" RVM evolution, SV-flavored

6.3 Verilog DW VIP workaround

This is a code sample of our OV+SV workaround via a direct Verilog instance.

```
`include "VmtDefines.inc"

`include "AhbMonitorDefines.inc"
`include "AhbMonitorMessages.inc"

`define AHB_MONITOR_0_INSTANCE ahb_monitor_0 // VIP to be ref'ed thru macro

// Point-to-point DW VIP usage workaround
wire [1:0] htrans_to_monitor_0 = ahbsvis[0].hsel ? ahbsvis[0].htrans : 2'b00;

ahb_monitor_vmt `AHB_MONITOR_0_INSTANCE (
  .hclk      (           SystemClock         ),
  .hresetn   (           hresetn             ),

  .haddr     ( {  32'b0, ahbsvis[0].haddr  } ),
  .hburst    (           ahbsvis[0].hburst   ),
  .hmastlock (           1'b0                ),
  .hprot     (           ahbsvis[0].hprot    ),
  .hrdata    ( { 992'b0, ahbsvis[0].hrdata } ),
  .hready    (           ahbsvis[0].hready   ),
  .hresp     (           ahbsvis[0].hresp    ),
  .hsize     (           ahbsvis[0].hsize    ),
  .htrans    (           htrans_to_monitor_0 ), // ahbsvis[0].htrans
  .hwdata    ( { 992'b0, ahbsvis[0].hwdata } ),
  .hwrite    (           ahbsvis[0].hwrite   ),
  .hsel      ( {  15'b0, ahbsvis[0].hsel   } ), // 16'h01 is actually enough

  .hgrant    (           16'h01              ),
  .hlock     (           16'h00              )
);

initial begin
  int tm_handle;
  int main_stream = 0;

  @( posedge SystemClock );

  `AHB_MONITOR_0_INSTANCE.new_ahb_transaction_monitor ( tm_handle,
    "XFER_DIR READ, WRITE; XFER_TYPE NSEQ, SEQ; XFER_SIZE SIZE32" );

  `AHB_MONITOR_0_INSTANCE.set_config_param ( main_stream,
    `DW_VIP_AMBA_AHB_LITE_PARAM,           1 );            // AHB-Lite
  `AHB_MONITOR_0_INSTANCE.set_config_param ( main_stream,
    `DW_VIP_AMBA_HDATA_WIDTH_PARAM,       32 );            // default
  `AHB_MONITOR_0_INSTANCE.set_config_param ( main_stream,
    `DW_VIP_AMBA_CHECK_PROTOCOL_PARAM,     0 );            // default
  `AHB_MONITOR_0_INSTANCE.set_config_param ( main_stream,
    `DW_VIP_AMBA_SLAVE_PRESENT_PARAM, 16'h01 );

  `AHB_MONITOR_0_INSTANCE.start;
end
```