Co-simulation with OSVVM

Simon Southwell

February 2023

Preface

The content of this document was originally posted on the OSVVM website as a blog. The original article can be found here and the text refers to the features released with OSVVM 2023.01.

Simon Southwell
Cambridge, UK
February 2023

Contents

Introduction

I have written previously about co-simulation in my series of LinkedIn articles about using the programming interfaces of logic simulator tools in a fairly generic way, summarising the possibilities and referencing, in outline, some real-world use cases. Since then, I have been collaborating to bring some of these co-simulation features to OSVVM, an open-source VHDL verification methodology consisting of a set of libraries and packages for easing and improving the verification of logic IP. In this article I want to take a look at some of the new features and how these might be used to extend the possibilities for verification with OSVVM to include developing and testing with software, interfacing to external C/C++ models, generating tests in C/C++ and more.

Co-simulation Definitions

Before we get going, I want to define what is meant by co-simulation in this context as it can mean different things to different people. At a base level it can just mean running logic simulations via secondary languages such as Python (e.g., cocotb). These can be very low level allowing the driving of individual signals via the secondary language to generate test vectors. At the other end I have seen co-simulation refer to running FPGA emulated logic hardware with the embedded software to provide a pre-silicon, system level, platform of integrated testing.

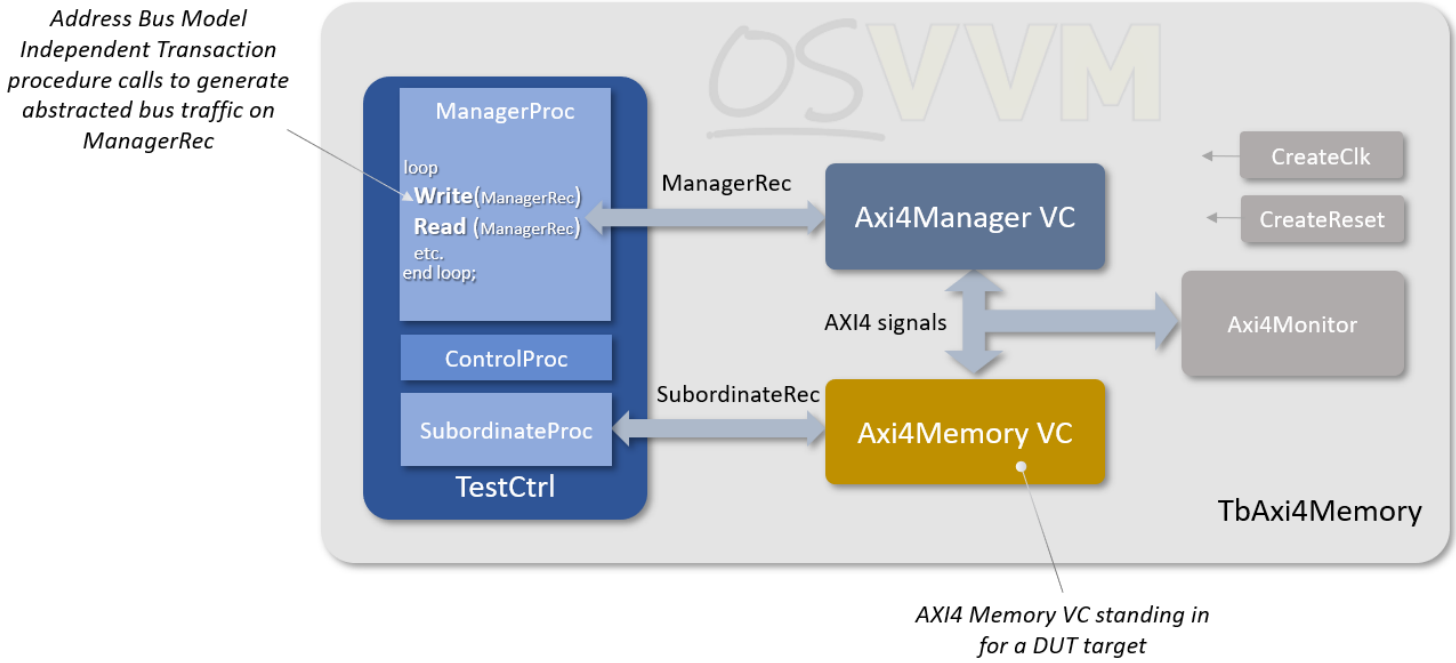
OSVVM is set of VHDL libraries and procedures to allow methodical testing of logic IP on a logic simulator to easily meet test and coverage goals etc. In this context, then, OSVVM co-simulation is also about running software in a logic simulation environment. The software side, as we shall see, is at the bus transaction level, utilising pre-existing OSVVM features, and can be anything from writing transaction test vectors in C++ to modelling a whole SoC system and its embedded software with a mixture of C++ models and real logic IP.

OSVVM Address Bus Transaction Level Modelling

There is not enough space to go through all of OSVVM's generous set of functionality here but, relevant to co-simulation, it includes Address Bus Model Independent Transaction features, whereby VHDL can issue abstracted bus read and write requests (either word or burst) via a set of API procedures that are then translated to specific bus protocols, such as AXI, via verification components (VCs)—so called Transaction Level Modelling (TLM). The advantage of this is that

tests that run for one protocol can be run for another protocol by just using a different VC to map the abstracted transactions to the particular protocol signalling.

A test bench within the OSVVM repository's AXI4 sub-module demonstrates how this works. The particular example we are going to reference is a test case located in `AXI4/Axi4/TestCases/TbAxi4_RandomReadWrite.vhd`, running on the test bench `AXI4/Axi4/testbench/TbAxi4Memory.vhd`. The block diagram below outlines this arrangement:



Here we see a manager process generating abstracted transactions via a record, `ManagerRec`, containing all the fundamental transaction information (e.g., address data, width, burst size etc.). In this example it does this with random calls to the OSVVM provided transaction procedures (such as `Write()`, `Read()`, `ReadCheck()` etc.) in a loop, terminating after a set number of transactions have been generated. The responses to the transactions, such as read data or error status, are returned within the same `ManagerRec` record back to the manager process. In this sense, the manager process here is acting like a processor core, as if issuing reads and writes on load and store instructions.

The `ManagerRec` is connected to an `Axi4Manager` verification component which converts the abstract transactions to specific AXI4 protocol manager signalling. A subordinate is connected to the AXI4 bus—in this example it is another OSVVM VC, being the `Axi4Memory`. Of course, in a real test environment it would be the device under test (DUT) that would be connected, which could be a single subordinate peripheral logic IP or a whole sub-system, with AXI interconnect and several IP

blocks. The use of the `Axi4Memory` VC in this example is just to demonstrate and test the OSVVM components and infrastructure but serves to have a target that read and write transactions may be directed towards. As such this `Axi4Memory` VC has a `SubordinateRec` output (of the same type as `ManagerRec`) which is directed at a subordinate process and allows cross checking between issued manager transactions and received subordinate transactions.

This, then, is a brief summary of one possibility for the use of the Address Bus Model Independent Transaction features of OSVVM. Much more detail can be found in the OSVVM documentation.

Adding Co-simulation to OSVVM

Using the basic outline from the previous section we would like to add co-simulation capability to OSVVM with minimal disruption to the transaction level modelling features that are already in place. In essence we still want to generate address bus transactions, but from software rather than VHDL. None-the-less there must be a connection point within VHDL and, ideally, the procedures provided for the transactions are replaced with a simple procedure that connects to the C/C++ software domain, hiding the details of this from the user. As such it would have, as for the address bus transaction procedures, a manager record argument for generating the abstracted bus transactions and receiving the responses. It would also need to have an interrupt input (more later) and some status outputs to flag errors within the software and to indicate when the software has completed. The definition for the procedure provided in OSVVM is shown below:

```
procedure CoSimTrans (
  signal   ManagerRec       : inout  AddressBusRecType ;
  variable Done             : inout  integer ;
  variable Error            : inout  integer ;
  variable IntReq           : in     integer := 0 ;
  variable NodeNum          : in     integer := 0
);
```

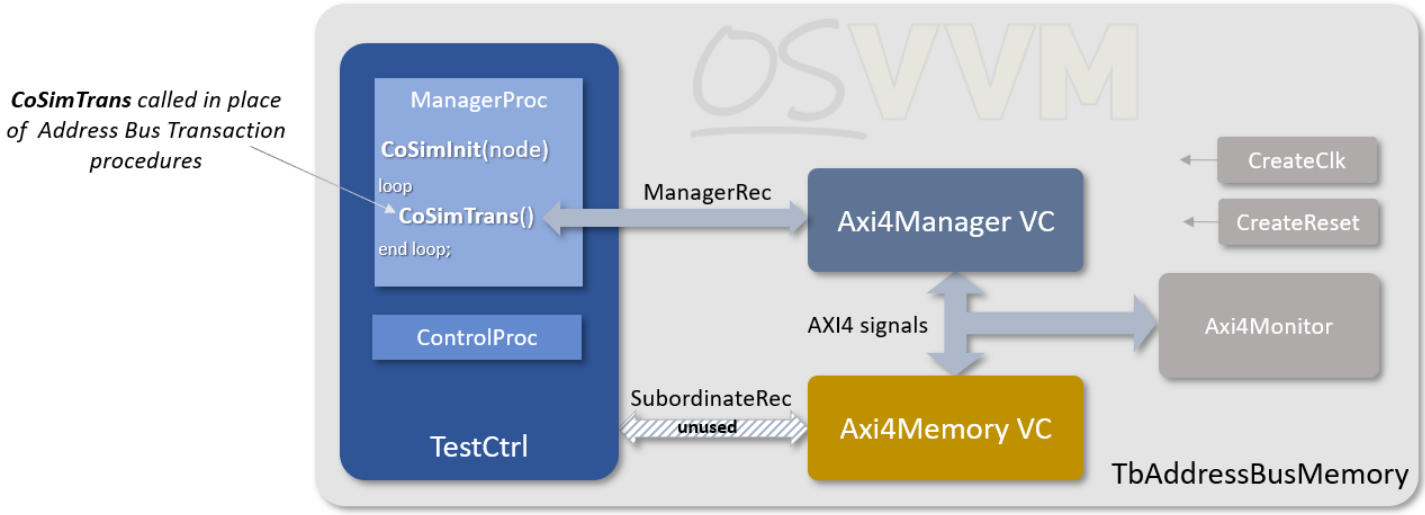
The arguments include those as discussed above, with a `ManagerRec` for transactions, an `IntReq` to receive interrupt state, an `Error` output, and a `Done` flag. The only additional argument is the node number (`NodeNum`). This allows multiple calls to the procedure from other processes for generating transactions on different abstracted busses. Each use of the procedure in a different process will use a unique node number. This is analogous to having multiple processor cores with their own

address bus generating reads and write transactions which can be connected, say, to an interconnect with multiple subordinate interfaces.

There is also an initialisation procedure which is used to initialise a co-simulation interface, starting up the software for that node, and must be called before any calls to `CoSimTrans` of the same node number.

```
procedure CoSimInit (variable NodeNum : in integer);
```

Taking the example that we discussed before, we can now modify this to use the new procedure.



Now, instead of calling `Read` and `Write` procedures in the loop, `CoSimTrans` is called instead. The rest of the test environment remains unchanged. The `SubordinateRec` is not used as the transactions now originate in software which does its own checks on data and a DUT that would sit in place of the `Axi4Memory` in a real test setup would also not have this. The above example outlines the test environment in the `CoSim` sub-module of the OSVVM repository and can be found in the file `CoSim/TestCases/TbAb_CoSim.vhd` which uses the test bench defined in the file `CoSim/testbench/TbAxi4/TbAddressBusMemory.vhd`.

And that's all there is to it from the VHDL side. But from this, as we shall see, we can do such things as run a program on a processor modelled in software right out of the `Axi4Memory` VC (standing in for DUT) or connect to an external program via a TCP/IP socket or just write tests in C++ to generate transactions, as well as much more. These are just some of the possibilities and the use of the co-simulation features are not limited to the arrangement discussed above or to the examples we will look at below—which is also true for the rest of the OSVVM features, and the

full range of OSVVM documentation should be inspected to understand all the possible use cases.

Having looked at the VHDL side of things, let's see what it looks like from the C++ software viewpoint and later we will look at how some of this is achieved by lifting the lid on the underlying co-simulation code.

The Software View

With `CoSimTrans` looping in a process we can now generate transaction from a C++ program. In a similar manner to `main()` being the entry point in a C program or `WinMain()` being the entry point in a graphical Windows program, the entry point for an OSVVM co-simulation program is `VUserMain`*n*`()` when *n* is the node number we used for the `CoSimInit` and `CoSimTrans` calls. Thus, if using the default value for `NodeNum` of 0, the software entry point is `VUserMain0()`. From here (or any other called function, class method or sub-program) we gain access to transactions via the provided `OsvvmCosim` class (defined in `CoSim/include/OsvvmCosim.h`). This is a wrapper class that sits on top of the lower-level co-simulation code and holds no state of its own except its node number, set at construction. This means it is safe to have as many instantiations of the class for a given node as needed without causing conflicts. When the class is constructed the node number is given as an argument to tie it to that particular node in VHDL. A test name can, optionally, be given at construction as well—usually only once for a given node if multiple instances of the class for the same node—and this identifies the particular software being run which may share the same VHDL test bench with multiple software tests.

The class provides methods to generate read and write transactions. These are summarised below:

```
uint<m>_t transWrite     (uint<n>_t addr, uint<m>_t  data);
void      transRead      (uint<n>_t addr, uint<m>_t *data);
void      transBurstWrite(uint<n>_t addr, uint8_t   *data, int bytesize);
void      transBurstRead (uint<n>_t addr, uint8_t   *data, int bytesize);
```

The methods are overloaded such that the size of the address type and the size of the data type determine the type of transfer. So, the `uint<n>_t` for the address can be `uint32_t` or `uint64_t` for 32-bit and 64-bit architectures. Similarly, the `uint<m>_t` type for the data can be `uint8_t`, `uint16_t`, `uint32_t` or `uint64_t` (if a 64-bit architecture) to allow byte, half-word, word and double-word transfers. For the burst methods, the address can be 32- or 64-bit, as before, with data transferred in or out of a buffer, passed in as a pointer, and the size of the burst via the `bytesize`

parameter. With these methods the program can now generate transactions in the testbench, just as if using the OSVVM Address Bus Model Independent Transaction procedures in VHDL.

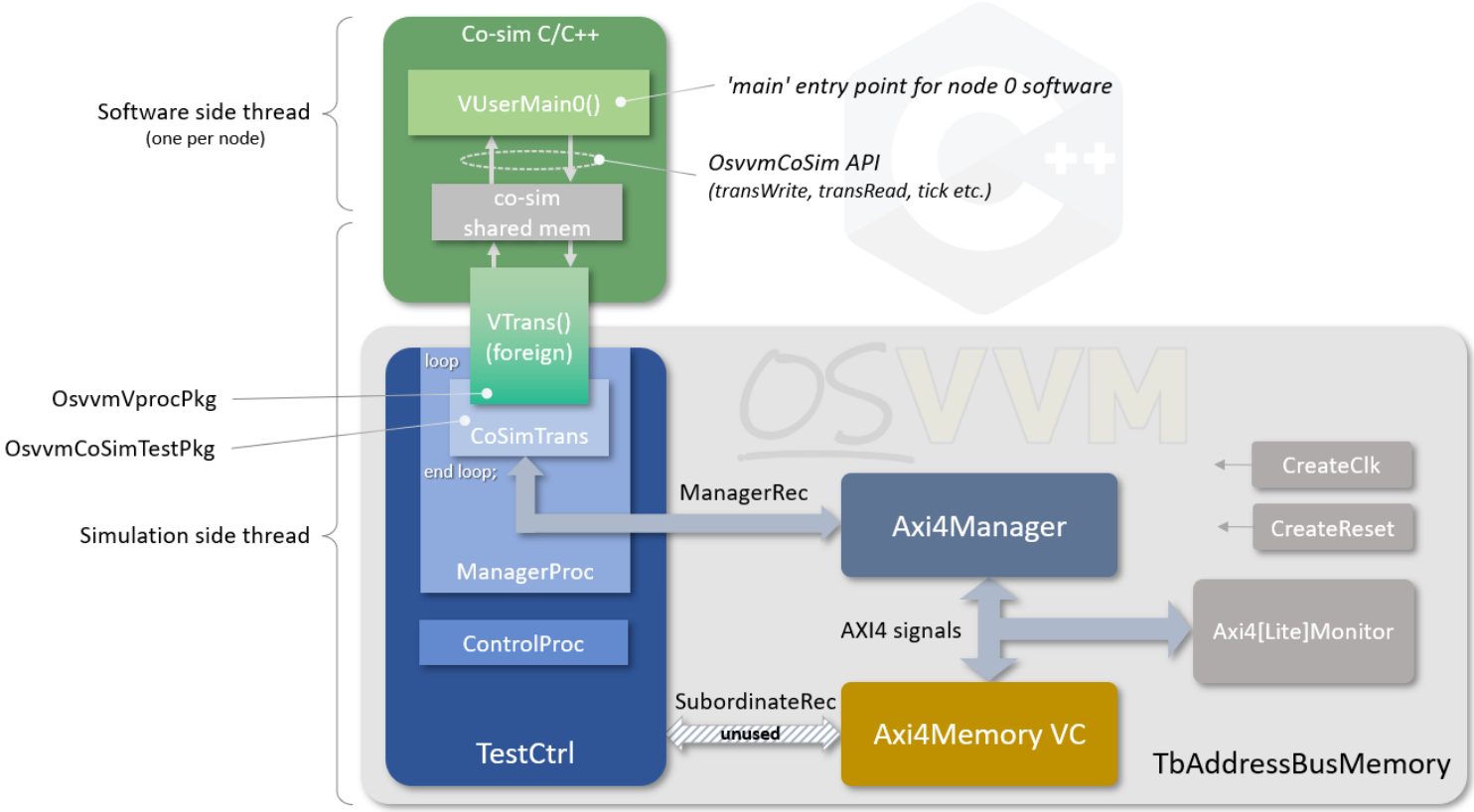
The way the underlying co-simulation code works, the simulation will not advance time and be blocked, once `CoSimTrans` is called, until a call to one of the transaction methods is made in software. The simulation will then advance for as many cycles as required to complete the transaction. Of course, the simulation code might do other things between calls to `CoSimTrans` that advances time, and the user software will be blocked in its call to the transaction method whilst this is going on. There may be times when one might want to advance simulation time without generating a transaction. The `OsvvmCosim` class provides a 'tick' method:

```
void tick (int ticks, bool done, bool error);
```

The term 'tick' here is loosely defined as basically a single call to `CoSimTrans`. This uses the OSVVM `WaitForClock` procedure on the `ManagerRec` parameter, and so will tick for the specified number of clock cycles associated with the transaction interface. Thus, the tick method allows for passing control back to the simulation for a set number of cycles when it is known that the software does not need to generate new transactions for some period. Notice, also, that there are 'done' and 'error' arguments. This is a means to affect the equivalent outputs on `CoSimTrans` to flag that the software is finished and indicate any error status.

The class also provides a method for registering an interrupt callback function, but we shall deal with interrupts in separate sections below.

Having looked at the VHDL side and now the software side, we have a system for effectively writing tests in C++ in place of calls to the VHDL procedures. Effectively we have made some of the Address Bus Model Independent Transaction procedures available in C++. This is useful in itself, but if that was all that could be done the use would be restrictive. At present only the blocking transaction functionality is available, though there are plans to add the other non-blocking, asynchronous and checking functionality as well, allowing a full range of tests to be written in C++. The diagram below summarises the situation with both the VHDL and C++ domains and an indication of the underlying code structure.



With this simple setup, beyond simply writing tests in C++, we can now do much more in terms of modelling SoC systems or connecting to external programs whilst running logic simulations with IP we are developing and wish to test. In the limit, the software used to drive the IP could also be run and co-developed with the logic IP using co-simulation, long before silicon becomes available. In the next couple of sections of this article I want to look at some examples that are available in the OSVVM repository of just such arrangements, but the usage is not limited to just these and it would be possible to hook up to other complex system models using these techniques. We will look at interrupts once we get to running a processor model and interrupting this.

Co-simulating with a Processor Model

In the discussions up until now, any code we write would be writing software to generate transactions for test vectors. Just such a program is in the `CoSim` sub-module repository in `CoSim/tests/usercode_size/VUserMain0.cpp` (there's also a burst equivalent in `usercode_burst`). In this section I want to look at using a C++ RISC-V instruction set simulator (ISS) model to hook-up to the co-simulation API so that it can access the logic simulated AXI address bus. This sits on top of the environment described in the sections above and requires no additional changes to the infrastructure already outlined.

At its most basic a single core processor has a clock and a reset, along with a manager type bus interface (I'm ignoring Harvard architecture for now) and one or more interrupt inputs. This is very similar to the `CoSimTrans` procedure's `ManagerRec` and `IntReq` parameters. The model being used is my riscV 32-bit RISC-V RV32GC ISS. This model pre-dates the OSVVM co-simulation features and has not been modified for OSVVM and I want to highlight how easy this was to integrate on top of the software API. The ISS allows for registering callback functions—one for memory accesses and one for interrupts. These features are used to generate read and write transactions and to allow program interrupts. Below is some abbreviated sample main code.

```
// Co-sim user code entry point for node 0
extern "C" void VUserMain0()
{
    OsvvmCosim cosim(node);

    rv32i_cfg_s cfg; // ISS config structure
    bool error = false;

    rv32* pCpu = new rv32(); // ISS object

    // Register memory access callback
    pCpu->register_ext_mem_callback(memcosim);

    // Register an interrupt callback with the CPU model
    pCpu->register_int_callback(interrupt);

    // Register an interrupt callback with the cosim software
    cosim.regInterruptCB(cosim_int_callback);

    // Load a program and run the ISS model
    if (!pCpu->read_elf("test.exe")) {
        pCpu->run(cfg);
        error = check_exit_status(pCpu);
    }
    else
        error = true;

    // Clean up
    delete pCpu;

    // Flag to sim that the test is finished
    cosim.tick(10, true, error);
    SLEEPFOREVER;
}
```

Here an `OsvvmCosim` object is created for node 0 (`cosim`). The ISS has a defined configuration structure of type `rv32i_cfg_s` and we will assume this is filled in appropriately. The ISS class itself is of type `rv32` and a pointer, `pCpu`, is set to a new

instance of the model. Two callback functions are then registered for memory accesses (`memcosim`) and interrupts (`interrupt`)—more in a bit—before loading a program (`pCpu->read_elf`) and running the model (`pCpu->run`). That's more or less it for the main program, with the interfacing to the co-simulation done via the callbacks. The memory access callback might look something like the following abbreviated code fragment:

```
int memcosim (const uint32_t byte_addr, uint32_t &data,
              const int      type,      const rv32i_time_t time)
{
  OsvvmCosim cosim(node);
  int        cycle_count = 5;
  uint8_t    rdata8;
  uint16_t   rdata16;
  uint32_t   rdata32;

  switch (type)
  {
    case MEM_WR_ACCESS_BYTE : cosim.transWrite(byte_addr,  (uint8_t)data); break;
    case MEM_WR_ACCESS_HWORD: cosim.transWrite(byte_addr, (uint16_t)data); break;
    case MEM_WR_ACCESS_WORD:  cosim.transWrite(byte_addr, (uint32_t)data); break;
    case MEM_WR_ACCESS_INSTR: cosim.transWrite(byte_addr, (uint32_t)data); break;
    case MEM_RD_ACCESS_BYTE:  cosim.transRead(byte_addr, &rdata8);  data=rdata8;  break;
    case MEM_RD_ACCESS_HWORD: cosim.transRead(byte_addr, &rdata16); data=rdata16; break;
    case MEM_RD_ACCESS_WORD:  cosim.transRead(byte_addr, &rdata32); data=rdata32; break;
    case MEM_RD_ACCESS_INSTR: cosim.transRead(byte_addr, &rdata32); data=rdata32; break;
    default: cycle_count = RV32I_EXT_MEM_NOT_PROCESSED; break
  }
  return cycle_count;
}
```

When the ISS calls the callback function it provides an address, a reference to a data word, an access type, and a current 'time' (which we will gloss over here as we don't really use it). Here we simply have a switch statement on the access types and call the appropriate `OsvvmCosim` method with the appropriate data type. Note that there are *instruction* write and read types as well as the normal data access types. This allows for loading of code (instruction writes) and reading of program instructions (instruction reads) via the co-simulation interface, as well as the normal load and store of data.

The interrupt callback is even simpler:

```
  uint32_t interrupt (const rv32i_time_t time, rv32i_time_t *wakeup_time)
  {
      *wakeup_time = 0;
      return IntReq ? 1 : 0;
  }
```

The ISS passes in the current time and a pointer for returning a 'wake-up' time—i.e., the time for the next scheduled call to the interrupt callback. Again, time modelling is of no interest and we will always set this to 0 to have no delay in calling. The callback returns 1 if an active interrupt is set, else it returns 0—the RISC-V architecture only defines a single external interrupt, with multiple interrupts handled by an external processor level interrupt controller (PLIC). `IntReq` is an external variable that is updated to reflect the current state of the `IntReq` parameter of the `CoSimTrans` VHDL procedure. In order to get hold of that we need to register another callback—this time with the co-simulation software. The `OsvvmCosim` class has a `regInterruptCB` method for registering a callback function which will be called whenever the `CoSimTrans` `IntReq` parameter changes. For our example, the code might look something like the following:

```
int cosim_int_callback(int int_vec)
{
    IntReq = int_vec & 1;
    return 0;
}
```

When the co-simulation software calls this it simply updates the `IntReq` local static variable with the low bit of the passed in interrupt vector state, making it available to the ISS via its `interrupt` callback. There are better ways of doing this than using a locally static variable, but things have been kept simple for this example.

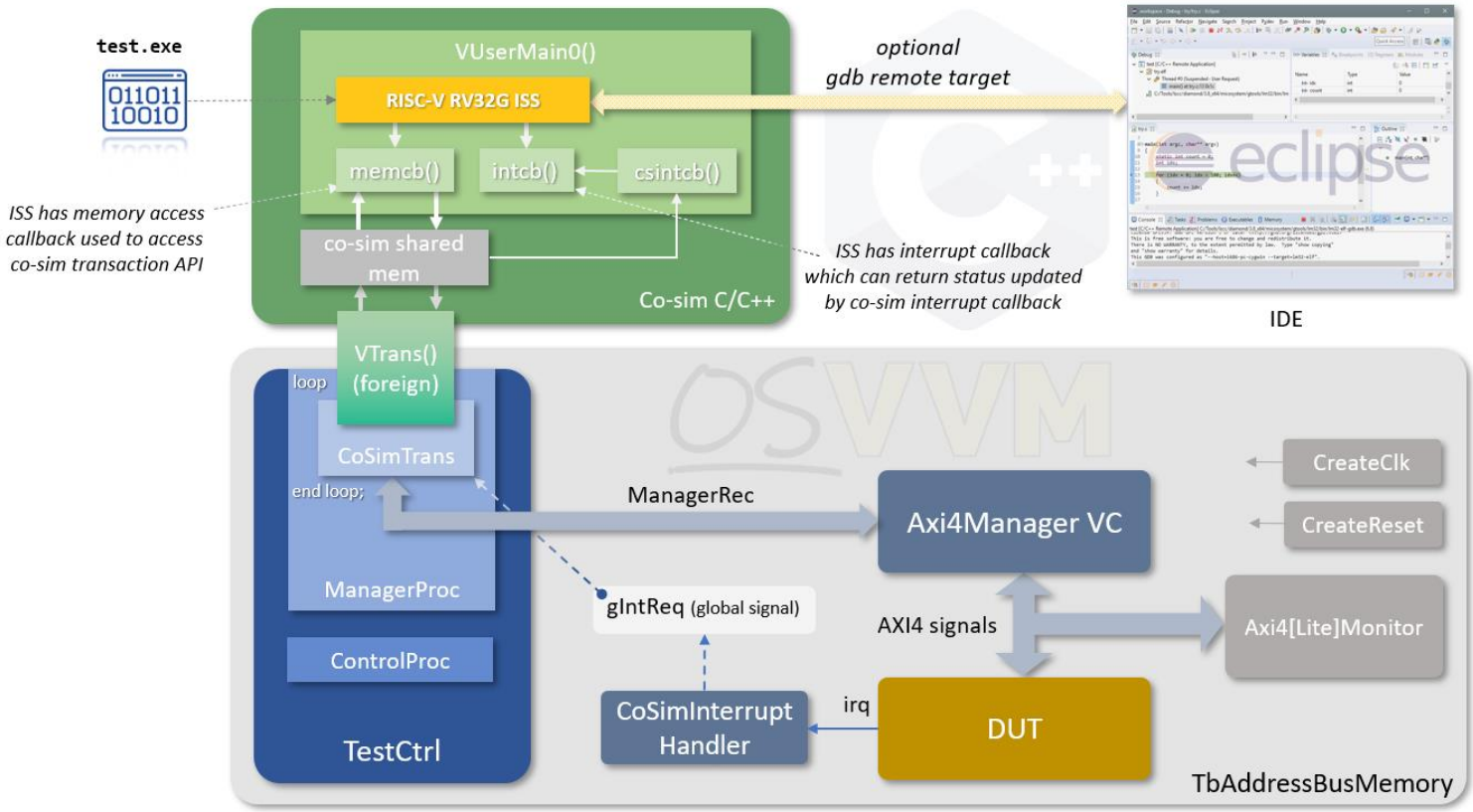
Note that the co-simulation callback function is only meant to update local state to allow the main user code to process it. It can't be used to make additional calls to transaction methods of an `OsvvmCosim` instance. This is because the callback is instigated from the simulation with the main program blocked on its last call to a transaction method. To call a new transaction method would be to do so in the middle of a pending transaction. The main code is responsible for modelling the actual interrupt actions, just as is done by the ISS model in this case.

So, having hooked up the ISS model to the co-simulation interface, what can we do with this. Well, the ISS can run cross-compiled RISC-V programs which we can load and run on the model. In the OSVVM provided example in `CoSim/tests/iss` a precompiled program called `test.exe` is used, alongside the `VUserMain0.cpp` program. This is actually a 3rd party piece of software. The RISC-V International organization provide a suite of instruction unit tests and the test code is one of these from the `rv32ui` set for testing byte store instructions. With the user code configuring the model to do run time disassembly and to dump the register state on termination the following shows a fragment of the output of the simulation:

```
00000564: 0x00a581a3    sb        a0, 3(a1)
%% Log    INFO      in manager_1,   Write Data.  WData: EFUUUUUU  WStrb: 1000  Operation# 451 at 21030 ns
%% Log    INFO      in manager_1,   Write Address.  AWAddr: 000015C3  AWProt: 000  Operation# 451 at 21030 ns
%% Log    INFO      in memory_1,   Memory Write.  AWAddr: 000015C3  AWProt: 000  WData: EFUUUUUU  WStrb: 1000  Operation# 450 at 21040 ns
%% Log    INFO      in manager_1,   Read Address.  ARAddr: 00000568  ARProt: 000  Operation# 532 at 21040 ns
%% Log    PASSED    in manager_1: WriteResponse Scoreboard,      Received: 0   Item Number: 451 at 21050 ns
%% Log    INFO      in memory_1,   Memory Read.  ARAddr: 00000568  ARProt: 000  RData: 02301063  Operation# 531 at 21050 ns
%% Log    PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 532 at 21060 ns
%% Log    INFO      in manager_1,   Read Data: 02301063  Read Address: 00000568  Prot: 0 at 21060 ns
00000568: 0x02301063    bne       zero, gp, 32
    *
%% Log    INFO      in manager_1,   Read Address.  ARAddr: 00000588  ARProt: 000  Operation# 533 at 21060 ns
%% Log    INFO      in memory_1,   Memory Read.  ARAddr: 00000588  ARProt: 000  RData: 0FF0000F  Operation# 532 at 21070 ns
%% Log    PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 533 at 21080 ns
%% Log    INFO      in manager_1,   Read Data: 0FF0000F  Read Address: 00000588  Prot: 0 at 21080 ns
00000588: 0x0ff0000f    fence     15, 15
%% Log    INFO      in manager_1,   Read Address.  ARAddr: 0000058C  ARProt: 000  Operation# 534 at 21080 ns
%% Log    INFO      in memory_1,   Memory Read.  ARAddr: 0000058C  ARProt: 000  RData: 00100193  Operation# 533 at 21090 ns
%% Log    PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 534 at 21100 ns
%% Log    INFO      in manager_1,   Read Data: 00100193  Read Address: 0000058C  Prot: 0 at 21100 ns
0000058c: 0x00100193    addi      gp, zero, 1
%% Log    INFO      in manager_1,   Read Address.  ARAddr: 00000590  ARProt: 000  Operation# 535 at 21100 ns
%% Log    INFO      in memory_1,   Memory Read.  ARAddr: 00000590  ARProt: 000  RData: 05D00893  Operation# 534 at 21110 ns
%% Log    PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 535 at 21120 ns
%% Log    INFO      in manager_1,   Read Data: 05D00893  Read Address: 00000590  Prot: 0 at 21120 ns
00000590: 0x05d00893    addi      a7, zero, 93
%% Log    INFO      in manager_1,   Read Address.  ARAddr: 00000594  ARProt: 000  Operation# 536 at 21120 ns
%% Log    INFO      in memory_1,   Memory Read.  ARAddr: 00000594  ARProt: 000  RData: 00000513  Operation# 535 at 21130 ns
%% Log    PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 536 at 21140 ns
%% Log    INFO      in manager_1,   Read Data: 00000513  Read Address: 00000594  Prot: 0 at 21140 ns
00000594: 0x00000513    addi      a0, zero, 0
%% Log    INFO      in manager_1,   Read Address.  ARAddr: 00000598  ARProt: 000  Operation# 537 at 21140 ns
%% Log    INFO      in memory_1,   Memory Read.  ARAddr: 00000598  ARProt: 000  RData: 00000073  Operation# 536 at 21150 ns
%% Log    PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 537 at 21160 ns
%% Log    INFO      in manager_1,   Read Data: 00000073  Read Address: 00000598  Prot: 0 at 21160 ns
00000598: 0x00000073    ecall
    *
PASS: exit code = 0x00000000 running test.exe

Register state:

  zero = 0x00000000   ra = 0x00000001   sp = 0x000015c0   gp = 0x00000001
    tp = 0x00000002   t0 = 0x00000002   t1 = 0x00000000   t2 = 0x00000001
    s0 = 0x00000000   s1 = 0x00000000   a0 = 0x00000000   a1 = 0x000015c0
    a2 = 0x00000000   a3 = 0x00000000   a4 = 0x00000001   a5 = 0x00000000
    a6 = 0x00000000   a7 = 0x0000005d   s2 = 0x00000000   s3 = 0x00000000
    s4 = 0x00000000   s5 = 0x00000000   s6 = 0x00000000   s7 = 0x00000000
    s8 = 0x00000000   s9 = 0x00000000  s10 = 0x00000000  s11 = 0x00000000
    t3 = 0x00000000   t4 = 0x00000000   t5 = 0x00000000   t6 = 0x00000000

%% DONE   PASSED   CoSim_iss  Passed: 988  Affirmations Checked: 988  at 21300 ns
simulation stopped @21300ns
Simulation Finish time 12:49:10, Elasped time: 0:00:13
Build Start time  12:48:51 GMT Mon Jan 23 2023
Build Finish time 12:49:10, Elasped time: 0:00:19
Build: Scripts_RunCoSimIssTests PASSED,  Passed: 1,  Failed: 0,  Skipped: 0,  Analyze Errors: 0,  Simulate Errors: 0
```

Here we see ISS disassembly output (coloured blue for clarity) alongside the logging from the OSVVM transactions and can see how the reading of instructions and store

of bytes corresponds to the transactions in the OSVVM simulation. So the RISC-V software that's running on the ISS model is actually running out of the OSVVM `Axi4Memory` VC in the logic simulation and doing loads and stores to this memory as well. The ISS also has remote gdb debugging capabilities and so can be connected to an IDE and RISC-V software debugged as for any other program, with stepping, breakpoints, variable inspection etc., all from the logic simulation. The diagram below summarises the situation described here:

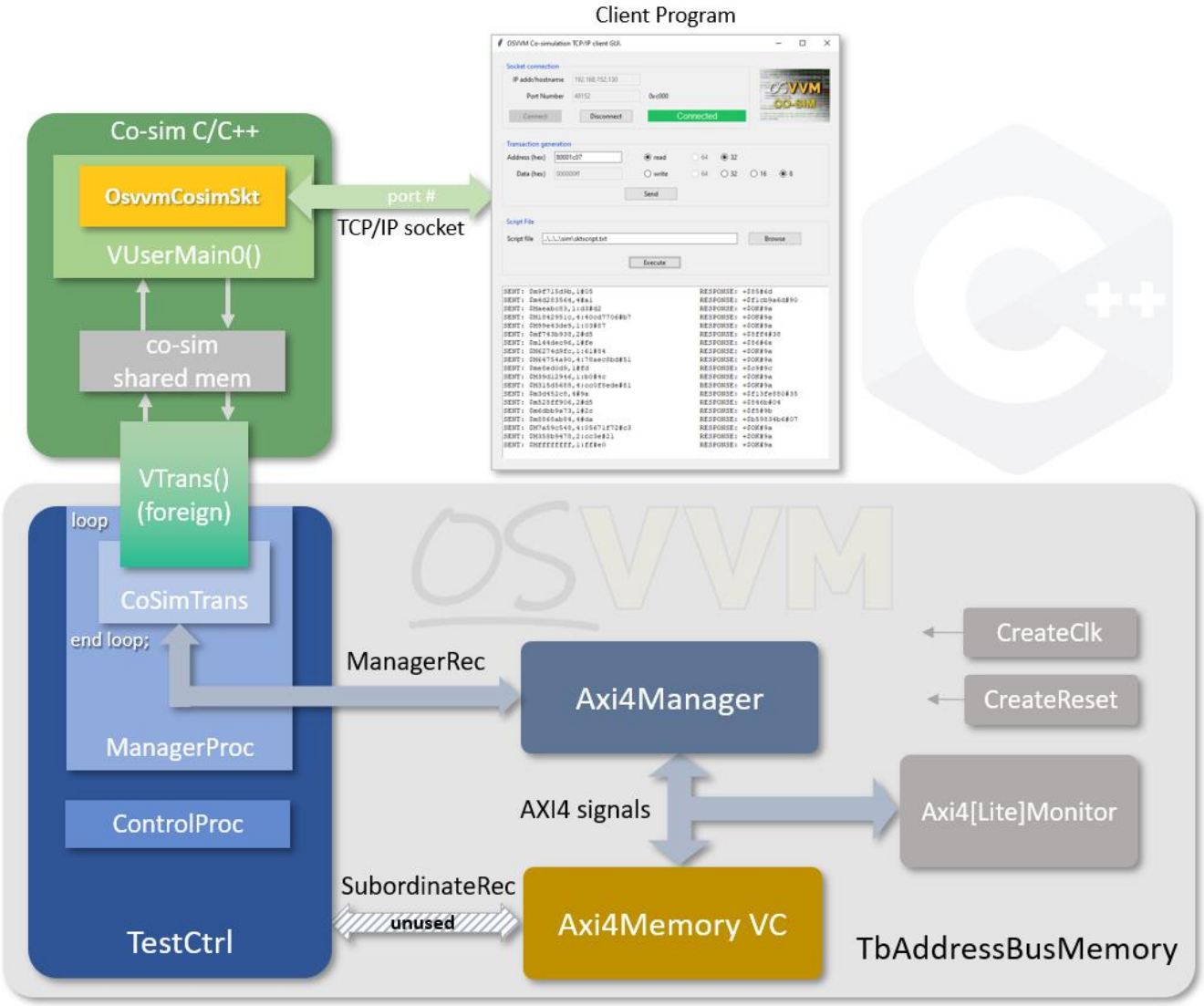


With any interrupt request status being passed into the `CoSimTrans` procedure calls the processor software can be interrupted (assuming its enabled). An example test for invoking an interrupt exists in the directory `CoSim/tests/interruptIss` using the `TbAddressBusMemory.vhd` test bench in `CoSim/testbench/`. Here the test can write to a set location in memory to invoke an interrupt request, allowing the RISC-V code to generate its own interrupt and check that it was indeed interrupted.

Hopefully it is not a giant conceptual leap to imagine a test environment where the `Axi4Memory` VC is replaced by some IP with memory mapped registers that can be accessed by software on the processor model to exercise its functionality, and that the code might even be the intended device driver for that block.

Connection to an External Program

The previous ISS example made the assumption that the system or processor model can be called from the `VUserMain0` code (or a sub-program of it). This might not be the case if the model we wish to use is a stand-alone executable such as a virtual machine (e.g., QEMU). Since we have a free hand in what code is written under the `VUserMain0` top level this can be anything we like, including inter-process communication software such as a TCP/IP socket, though any IPC method can be utilised. A simple socket class is provided in OSVVM as an example, located in `CoSim/code/OsvvmCosimSkt.cpp`. This allows remote connection via a TCP port and accepts communication that loosely follows the gdb remote protocol for reading and writing to memory. Sending read and write commands over the socket instigates transaction reads and writes on OSVVM, returning any read data responses as responses to the commands. The protocol does not yet support bursts or interrupts but could be extended to do so. It is meant as a demonstration of external connection rather than be a fully functional solution, as the protocols and requirements will vary wildly between real-world use cases. This use case, then might look something like the following.



This arrangement uses a python program as a stand in for an external program with a socket interface. The test code is in `CoSim/tests/socket` with the python GUI program in `CoSim/Scripts/client_gui.py`. Essentially, though, this is the same arrangement as for any of the previously discussed examples—that is, some software makes calls to the C++ co-simulation API to generate transactions in the simulation that get translated from abstract accesses to bus specific protocols, such as AXI4. The same test could run on a different memory mapped bus protocols (e.g. Avalon) without modification. Whether the software is running a processor system modelled in C++ or being controlled via a TCP/IP socket connection to an external program, the simulation traffic is the same in nature. Also, because the traffic is generated in OSVVM, even if originating from software, all the advantages of OSVVM are available, with all the metrics, logs, and results that that brings.

Under the Hood

There's no space to go into full detail here about how the co-simulation software interfaces with a logic simulator such that a software program can appear to free run whilst interacting with a logic simulation that also appears to free run, but it will be instructive to give an overview to help understand what's going on and inform on efficient usage. This is really for those interested in the inner workings or those intending to interface with more complex software models, so feel free to skip this section if you would just be a user of the features.

To get to the end of the story first, it's all done with threads and mechanisms put in place to ensure that, actually, the threads aren't running concurrently but only one is ever unblocked at a time.

The `VUserMain`*n* entry points (as many as there are nodes) are each running in a separate thread from the simulator (which is the 'main' thread, if you like). These were set up and instigated when `CoSimInit` was called in VHDL for that node. These all need to be coordinated to exchange information between the C/C++ and logic simulation domains. When the simulation calls the `CoSimTrans` procedure for a given node it will block on that call. The corresponding `VUserMain` thread will free run until it makes a call to an `OsvvmCosim` transaction or tick method for the same node. The transaction information is loaded to a 'send' structure (one for each node) and the simulation call is unblocked, with the `OsvvmCosim` method now becoming blocked. The simulation side software now accesses the transaction structure and returns to `CoSimTrans` which generates the transaction in the simulation, advancing time, and at some point, makes another call to the `CoSimTrans` procedure again with any response information. The response is copied to a 'response' structure and the originally called `OsvvmCosim` method is unblocked and can access this and return the data and/or status. The `CoSimTrans` call is blocked once more, waiting for the next time a transaction or tick method is called from software (for that node). This happens for as many nodes that are used in the simulation and gives the illusion that software and simulation are all free running programs.

The details aren't important, but this is all done using node specific send and response structures as shared memory between the user threads and the simulation thread and synchronised with semaphores. The sequence described above ensures that only one thread is ever unblocked at any one time making the access of shared data perfectly safe. This is why, with the interrupt callback, it is perfectly safe to update state in a `VUserMain` program without risk of race conditions, as the callback is called from the simulation thread and thus the `VUserMain` thread will be blocked

and cannot access the updated data until it is unblocked when the simulation calls `CoSimTrans` once more. It is important to note that between calls to `OsvvmCosim` transaction or tick methods simulation time is not advancing and the software is effectively running at infinite speed (as far as the logic simulation is concerned). This is not usually a concern, but if the simulation does need to advance between certain points in software execution the `OsvvmCosim` tick method can advance time without generating new transactions, and if the calls to `CosimTrans` are set up so that it is called at the clock rate whilst ticking, then time can be advanced accurately to the resolution of the clock rate.

Having explained the use of threads in the co-simulation workings and how they are, in fact, not free running, I want to mention that the software does allow multi-threaded programs to run and access the `OsvvmCosim` methods, and effectively share a given node's transaction interface. At the heart of the co-simulation software is an 'exchange' function which synchronises the threads and copies the send and response data in and out of the shared structures. This function is protected by mutexes on a per-node basis. Therefore, any threads using the same node will have atomic exchange of data to instigate a transaction (or tick) without fear of another thread stomping over this exchange.

There are probably a dozen ways the same functionality could be achieved using more modern libraries and methods. The PLI side of the software must be C (at least have C linkage), and so this simulation side code was chosen to be written in C. The origins of this technology also go back 20 years, and so some methods have probably been superseded since then. At any rate, this is all hidden from the user behind the VHDL `CoSimTrans` procedure and `OsvvmCosim` class methods, but for anyone interested in looking at the source code to see how it works, this gives a flavour of what you'll find.

Before we finish this section, with all this talk of threads, wouldn't it be easier if we could just call the simulation as if it were a function call and then we wouldn't need to muck about with threads. Most simulators that I have used do not provide any such feature, as the simulator is a free-standing executable rather than a library that can be linked to a user program. One exception though is GHDL. This provides a method for calling GHDL via a `ghdl main()` function. GHDL comes in various flavours and the 'mcode' version does not support this. However, the others do and the OSVVM repository (under the `CoSim` sub-module) provides a demonstration test in `CoSim/tests/ghdl_main`. Here the user VHDL code, compiled into by GHDL into object files, are gathered into a library which can be linked with user code along

with some GHDL libraries, into an executable. From the user code, `ghdl_main()` can be called to start the simulation. Unfortunately, the `ghdl_main()` call is blocking until the simulation terminates, so calling the `OsvvmCosim` methods isn't possible from the same thread—so the call is made in a separate thread! There's no getting away from it, I'm afraid. None-the-less, this is still useful as this can be linked with code that extends or forms part of another modelling system, such as QEMU, which is, itself, a stand-alone executable. It is a pity that other simulators don't have this feature. This method is not strictly how the use of `ghdl_main()` is documented, as it is still expected that user code will be provided as a shared object to the simulation executable for dynamic loading. The OSVVM test compilation simply does the final link to allow the user code to be part of the executable that GHDL generates.

Conclusions

We have looked at the new co-simulation features of OSVVM and how they can be used to extend verification into the C++ domain with, ultimately, the ability to run software interacting on a system model with logic design on a simulator via the OSVVM TLM features. We looked at a couple of examples of using a RISC-V processor model and of connecting to an external program to drive transaction generation. The features are not limited to these examples and one can extend to C++/logic co-simulation to any level of sophistication. We also had an overview of how the co-simulation software works.

The emphasis on the OSVVM co-simulation features is to expose parts of the transaction layer modelling features of OSVVM in the C++ domain, such that a simple class gives access to those features from a user written C++ program. From that point it has been demonstrated that this opens up having logic simulation as an extension of software models of any complexity and of executing embedded software driving logic IP in one, pre-silicon, environment. This is seen as complementary to other techniques, such as FPGA emulation, but gives the ability of early co-development of logic and driver software, as well as a host of other possibilities.

What's Next

The current release of OSVVM gives access to a sub-set (though a useful one) of the possible Address Bus Model Independent Transactions procedures. These are enough to generate transactions from bytes to burst accesses as atomic actions. It is planned to add support for the split transactions as well as for the checking flavours

of reads etc. This will make it easier to write complete tests in the C++ domain as would be available from VHDL.

OSVVM also has model independent streaming functionality as well as address bus, and it is also planned to add support for this in co-simulation to give greater flexibility in system modelling with the co-simulation features. So watch out for these updates coming soon.