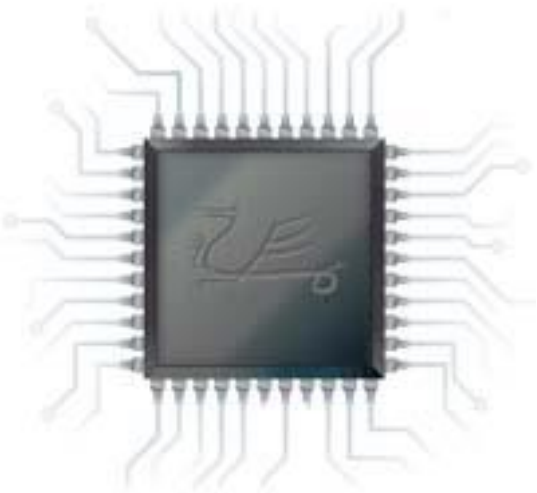
# Using Multiple Transaction Nodes in OSVVM Co-simulation



Simon Southwell

June 2023

Simon Southwell
Cambridge, UK
June 2023

# Contents

# Introduction

The OSVVM co-simulation environment has the ability to generate transactions from C++ to drive various virtual components (VCs), such as AxiBus, AxiStream, Ethernet, Uart, dual port RAM etc. But did you know that you can drive multiple VCs from within the same test bench, all driven by C++ software? To do this we have the concept of 'nodes'.

The OSVVM co-simulation documentation does document the concept of a node, and a unique node number for each co-simulation instantiation, though in a reference manual manner. In this article, though, I want to discuss in more detail what the nodes are, what they are used for, how to instantiate them in a test bench, and how the software for each node, in a multi-node system, can communicate and synchronise with each other and even be multi-threaded. We'll close up by taking a look at some ideas on how these facilities can be used to model embedded software architectures.

# What is a Co-simulation Node?

In short, a co-simulation node is a source of transactions that has a unique node number. The OSVVM library provides a set of VHDL co-simulation procedures that can be called from a process to drive an OSVVM abstracted bus independent transaction interface signal. It's these interface signals that drive the verification components (VCs) which convert the high level, abstracted, transaction information into specific protocol signalling, such as AXI4 or Ethernet. As of OSVVM release 2023.05 the co-simulation VHDL procedures available are as listed below:

- `CoSimTrans:` manager transactor driving an `AddressBusRecType` signal
- `CoSimResp:` subordinate responder driving an `AddressBusRecType` signal
- `CoSimStream:` streamer driving `StreamRecType` signals

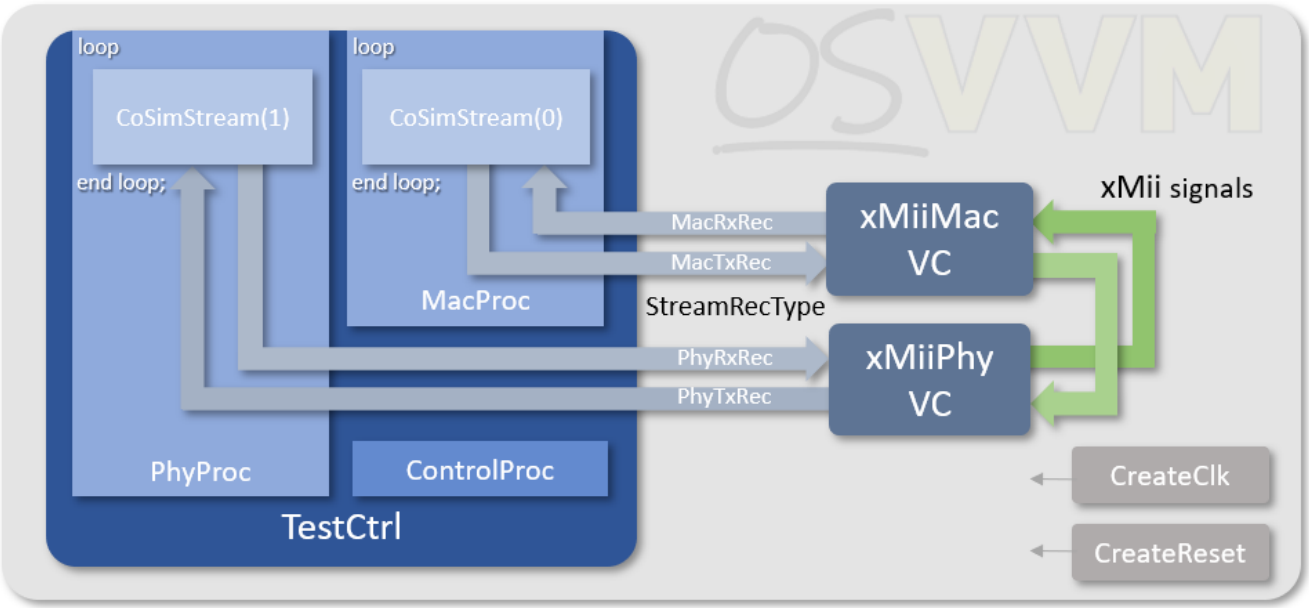
Each call to one of these procedures will generate either a new transaction or allow simulation time to tick by (or some other utility functions, which aren't relevant to this discussion).

One or more of these procedures can be called from within its own process, usually in a loop, to drive multiple bus model independent transaction signals attached to multiple VCs. Each co-simulation procedure used will have a unique node number

(more later), and all can be the same procedure type, or a mix of any of the types, as appropriate to the test environment being constructed.

## What can Multiple Nodes be Used For?

A couple of scenarios can illustrate how a multiple node environment could be used. For example, one might have two `CoSimTrans` co-simulation procedures driving two AXI4 VCs if, say, testing an AXI4 interconnect as the DUT, where the two `CoSimTrans` procedures are generating transactions, acting as if they are two processor cores. In another scenario, there might be one `CoSimTrans` and another `CoSimStream`, where the DUT is an Ethernet MAC, attached to an AXI bus VC to access its memory mapped registers, whilst the other is a `CoSimStream` connecting to an `EthernetStream` VC to receive and respond to the IP's traffic. You get the idea—there is no restriction on the mix of types. There is a slight restriction in that the default maximum number of nodes is 64, but even this can be overridden at compile time if necessary. The diagram below shows a multi-node set up for one of the OSVVM co-simulation tests—in this case for driving ethernet VCs.




For reference, this example is found in the files in `CoSim/testbench/TbEthernet`, and `TestCtrl` in `CoSim/testbench/TestCases_Ethernet`. The `TestCtrl` architecture, where the `CoSimStream` procedures are called, is defined in `TbStandAlone.vhd`.

## Node Number

The node number is critical in delineating the various calls to OSVVM co-simulation procedures. All of the OSVVM procedures take a `NodeNum` argument, and this connects the calls to that procedure with the user software that is driving the generation of transactions. Each procedure that drives a given model independent signal (or signals, if the connected VC requires them) must have a unique node number, and must also be consistent between calls to that procedure—no dynamically changing the `NodeNum` between calls. This ensures that the user software running for a given node is the only source for transactions to a particular VC.

### Initialisation

The transaction procedures connect user software for a given node number to a particular program. If a transaction procedure uses a node number of 0, then the user is expected to supply a C function (or one with C linkage, if compiled in C++) that is called `VUserMain0()`. Likewise, a node number of 1 requires a function called `VUserMain1()`, and so on for each node number used. You can think of these as the `main()` program running on a 'virtual processor' core, with a separate core for each node. Access to generating transactions is provided with one of the API classes, such as `OsvvmTrans` or `OsvvmStream` (more later). At construction, these are assigned the matching node number to link it to the VHDL co-simulation procedure.

These user programs, however, must be executed to start generating transactions, and so an OSVVM co-simulation procedure, `VInit(NodeNum)`, is provided to do just that. It takes a single node number argument, does some internal initialisation, and will start the user code for that node running. It *must* be called before any call to a co-simulation transaction procedure is called that uses that same node number, or the simulation is likely to hang. In addition, if it is called with a node number that does not have a matching `VUserMain`*`N`* program, then an error is likely to occur.

## User Node Programs

The user programs, as we established in the last section, have a given entry point for each node but, from then on, they can have any complexity of hierarchy, with sub-routines, classes etc. to organise the flow of the program.

A set of API classes are provided to match the type of transactions being used by the co-simulation VHDL procedures to drive a VC. The current classes supported as of release 2023.05 are:

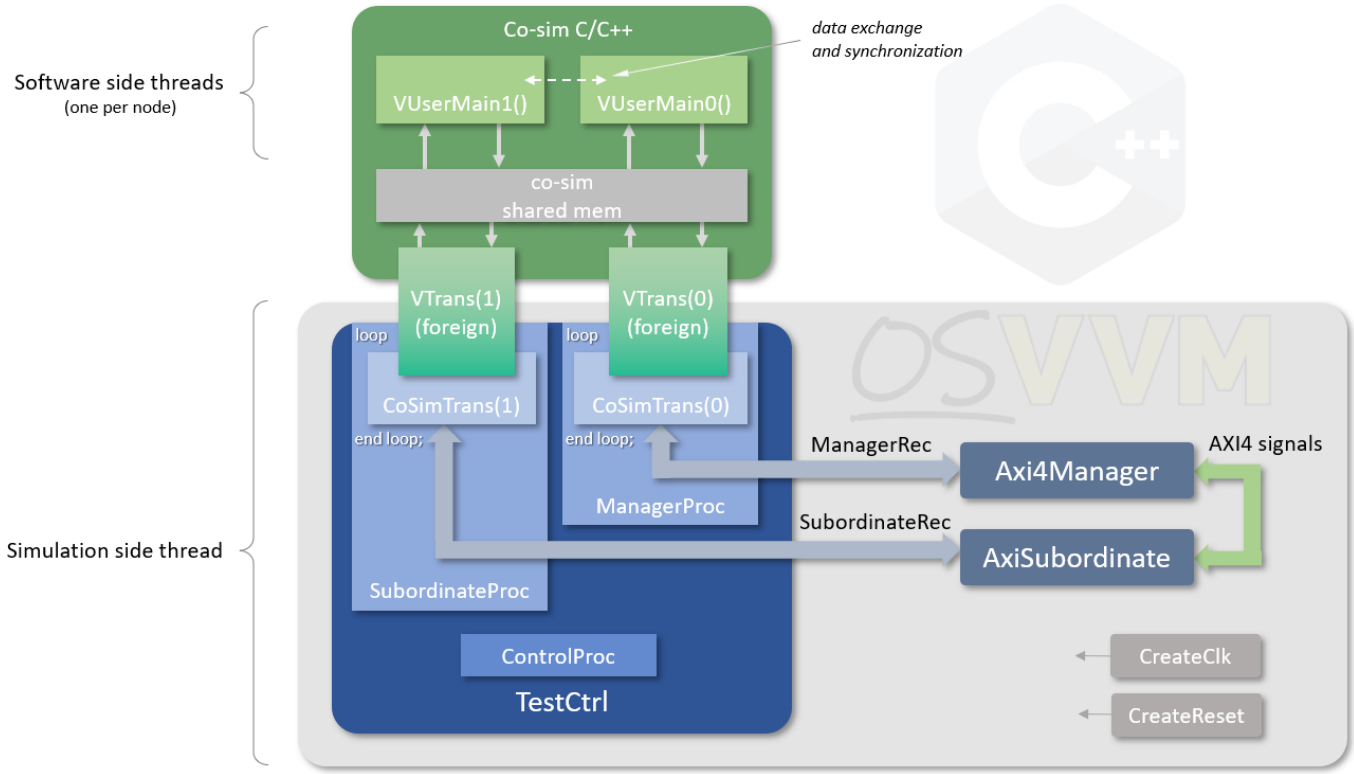
- `OsvvmTrans:` manager transactor API class
    - `OsvvmTransInt:` manager transactor API class with nested interrupts
- `OsvvmResp:` Subordinate responder API class
- `OsvvmStream:` Stream API class

Use of these classes is documented elsewhere (see here and here), so I won't go into details but, suffice it to say, the user code generates transactions with calls to the methods provided by these classes, as well as means to advance simulation time without generating a transaction.

So, we have multiple nodes, each running a user program, as if running on its own processor. How?

## Threads Under the Hood

Relevant to our discussion, it is worth having a look at what's going on under the hood that allows us to have multiple running user programs. Without going into a lot of detail, each user program is running in a separate thread—and these threads are separate from the simulator executable's 'thread', which is just its main program flow. (I'm sure it will have multiple threads within its execution but, as an executable, it has a single 'main' entry point, just like our co-simulation user programs.) The OSVVM co-simulation takes care of managing the exchange of information between the user programs and the simulation and keeping this all synchronised. The diagram below summarises the setup for one of the tests in the OSVVM co-simulation test suite:




For reference the files are in `CoSim/testbench/TbAxi4_Responder`, with the `TestCtrl` architecture file in `CoSim/testbench/TestCases/TbAb_Responder.vhd`.

Here we see two processes calling `CoSimTrans` procedures with a node number of 0 for the manager and a node number 1 for the subordinate. Each calls an underlying foreign procedure (`VTrans`) with the node number to connect to the C/C++ domain. The OSVVM co-simulation software manages the exchange of this information via some shared structures in memory (one set for each node). As shown, each of the `VUserMain`*N* functions are running in a separate thread, which are separate from the simulator program main program thread.

## Communication Between User Programs

Now it happens that the synchronisation mechanisms that have been put in place in the OSVVM co-simulation software (not coincidently) ensure only one of the user threads or the simulation are running at any given time, whilst the others are blocked. As well as ensuring safe communication between the user program threads and the simulation, it also has the benefit in that it allows us to have safe communication and synchronisation between the user programs as well, and without the need to use thread synchronisation methods in the user code. There are

some things we must take care of, but these are just programming requirements, and not calls to OS features.

## Exchanging data

In a normal multi-threaded program, care must be taken when exchanging information between these free running threads. If data from one thread is only partly constructed when the thread is de-scheduled and the receiver thread is activated, the program can fail to act as intended. This is still true of the co-simulation user code threads—accept now that has already been done by the co-simulation software. Therefore, any data can be constructed for reading by another user program safely. Not until a new call to a transaction API class method is made by the user program that's generating the data for use by another user program, will that user thread be de-scheduled, and the target program have a chance to be activated and read the data.

The method of data exchange is entirely a choice of the programmer, be it a single variable, or a complex data structure. The main point is that its generation is 'atomic' and requires no further action by the programmer. An example use of such an exchange comes from the example of the diagram above, where there is a transactor and responder thread. Here it is useful to send information of the data it used to generate a transaction directly to the responder thread which can then check the validity of the data it actually received.

## Synchronisation

As well as exchanging data between user programs, it is useful if they can be synchronised to co-ordinate that data exchange. For instance, taking the example from above, the responder software might want to control when the transactor software sends the next set of data so that a send-receive-check scenario is established for each individual test point. Without any synchronization, the transactor thread will simply run through its program, sending transactions at arbitrary times relative to the responder.

Since we already know that data exchange is safe between user programs, we can use a data variable as a 'barrier'. A simple integer variable, initialised to 0, can be used as such a 'barrier'. The responder increments the variable whenever it wants the transactor to advance to its next transaction generation. The transactor 'waits' on the barrier variable to increment before advancing. This is the situation used in

the test program in `CoSim/tests/responder.cpp` that runs on the test environment as shown in the diagram above.

So, what do we mean by 'wait'. As mentioned in the co-simulation documentation, a user program cannot loop internally on some state that it relies on from the simulation, otherwise the simulation will hang as no simulation time passes. Given that other user programs will, therefore, not be advanced, waiting on any state change from them will also cause a hang if simulation time is not advanced. In our example for transactor/responder, the transactor can loop, waiting on a barrier variable, so long as it allows simulation time to advance. The simplest way to do that is to tick for a cycle at each iteration of the loop. This is what the test code does via a local function:

```
extern int barrier;
static int last_barrier = 0;

static void waitOnBarrier(void)
{
    OsvvmCosim  cosim(node);

    while (barrier <= last_barrier)
        cosim.tick(1);

    last_barrier = barrier;
}
```

This is the simplest form of sychronisation. A dual synchronization can be achieved using the barrier variable for 'acknowledging'. For example, if the code waiting on a barrier is released when incremented, it can do some processing and then decrement the barrier variable to indicate it has completed an action. The barrier incrementing code can then wait on the barrier returning to zero to know that the other user program has completed some process. Thus, the two programs can be locked-stepped.

The barrier variable works a lot like a semaphore, and this is because the user programs are synchronised on semaphores under the hood. Therefore, any action that can be done with a semaphore can be emulated with this kind of use of barrier variables between the separate nodes' programs.

## Using Multiple Threads within a Single Node

Up until now, although, as we've seen, each user program is actually running in a separate thread, we have assumed that each user program for a given node is a single thread of execution. It may have any level of hierarchy, but the assumption has been it is a single thread. Is this a limitation required by the co-simulation environment?

Actually, no. The co-simulation layer provides for the situation where the software running on a given node is driven by multiple threads. The handling of this might have been passed on to the user code which would then have to take steps to manage the co-simulation API interface as a shared resource, but this is taken care of within the co-simulation software layer. At the heart of the co-simulation software is an exchange function that manages the data between user code and the logic simulation. At the point of exchange, where a user thread is blocked and the logic simulation is released, the code is guarded using mutexes so that it becomes an atomic operation. There is a mutex setup for each node so that each node does not interfere with other nodes program flow but, within a node, access to the co-simulation features, via the API classes, is thread safe, and each thread can drive the interface without further coding requirements.

A use case for such a requirement might be that the code running on a node is retargeted multi-threaded embedded software, cross-compiled for the co-simulation platform, making read and write accesses over a memory mapped bus. With memory access to certain address regions intercepted and sent to the simulation via the APIs, a true co-simulation environment is achieved with embedded software co-simulating with its target logic hardware in simulation. Since the embedded software can be multi-threaded, the APIs need to be thread safe—and they are.

## Accessing Different Nodes from a Single Program

We've already seen that we can drive different transaction interfaces using separate nodes running separate user programs, but it is still possible to have multiple threads from a single program that accesses different transaction interfaces, each on a unique node. This more closely matches a muti-threaded software environment where there is a single main program that spawns multiple threads.

In order to drive a second node a `CoSimInit` call must still be made in VHDL for this additional node (just as for the simpler cases), and this will require an equivalent

VUserMain*N* function. Now, though, the user function has very little to do and just needs to raise a flag that it has been called (see the discussion on barriers above) and then it can exit. A scenario might be that from within two separate VHDL processes, CoSimInit is called for each node, just as for ordinary cases. Let's say that we are using node numbers 0 and 1, and that the code running for node 0 is to be the main code. When VUserMain0 is called, it will do its initialisations and then start a new thread (with, for example, a function called RxDriver) for node 1's transaction interface. This new thread must now wait on a barrier. At some point (which could be before or after VUserMain0 is called), VUserMain1 is called. All this must do now is increment the barrier and then it can return. The RxDriver code running in the new thread is released and can then generate transactions for that node's transaction interface using an API class with a matching node number. Meanwhile, once VUserMain0 has spawned a node 1 thread, it is free to generate transactions on the node 0 interface, perhaps by calling a sub-program TxDriver. Note that the two driver functions, unlike having separate node programs, are free running threads, so both the RxDriver code (in our example) and the TxDriver code, if they need to share data, must follow all the thread safe precautions, and the co-simulation software layer will not take care of this in this scenario. Therefore, this setup is seen as an advanced configuration for allowing modelling of, say, real-time multiple threaded embedded software, and to be used by those who are confident in its safe implementation.

An example use for this scenario might be driving a stream bus model independent interface connected to an AxiStream or Ethernet VC. The CoSimStream co-simulation VHDL procedure allows for connection to both a transmitter and receiver StreamRecType signal, and this is ideal for the previous case where we had a single node driving Tx and Rx with different threads within the node's user program. With this new configuration, though, the CoSimStream procedure can be called (in a loop) from separate processes, with different node numbers, where one of the interfaces (either the Tx or Rx) is connected to a dummy signal. Now, the TxDriver thread can drive a TX interface of one node, and the RxDriver thread can drive the RX interface of the other node. Again, this can be achieved with separate user programs, called from, say, VUserMain0 and VUserMain1, but this configuration maps to a single multi-threaded program to allow easier mapping to real-time embedded software architecture with simple setup and initialisation as described. The usual API access class for the software is via the OsvvmCosimStream class, which provides both Tx and Rx methods. However, if now separating these functions, then one of two supplied

derived classes can be use as appropriate (either `OsvvmCosimStreamTx` or `OsvvmCosimStreamRx`) which limit the available methods to only the relevant functionality to prevent accidental calling of the inappropriate methods which would have undefined results.

## Using a Separate Node for Interrupts

OSVVM co-simulation provides for modelling interrupts (see the blog on interrupts for details), with the `CoSimTrans` procedure having an `IntReq` input argument, and the C++ API of the `OsvvmCosim` class providing a `regInterruptCB` method which allows a user program to register an interrupt callback function. This is called whenever the `IntReq` argument of a call to `CoSimTrans` changes state, passing in the new `IntReq` value to the user's callback function. In the interrupts blog it was mentioned about interrupt resolution and the need for not 'ticking' for extended periods of time (some of which is mitigated by the use of the `OsvvmCosimInt` class, described in the blog). The interrupts are not lost, but the latency before the code can handle the change in interrupt state might be artificially extended if care isn't taken.

One way around this is to have a separate `CoSimTrans` in its own process with a unique node number, wiring off the transaction signal and control outputs, and simply using the `IntReq` and `NodeNum` inputs. However, a convenience procedure is also provided, called `CoSimIrq`, with just the two `IntReq` and `NodeNum` inputs. Since this is not connected to transaction signal it has no concept of time and calls to a tick method in the C++ API do not influence the advancement of the simulation, and so would be used in a loop (just as for `CoSimTrans` or any of the co-simulation procedure calls), but with time advanced externally within that operational loop. This is done so that just when the procedure is called is completely up to the implementation, and also so that it *cannot* be blocked with a 'tick' call with a long delay. This could be by simply having a "`WaitForClock(ModelIndependentRec, 1)`" call in the loop, to using a "`wait on IntReq`" to only call `CoSimIrq` when the interrupt signal changes or, indeed, whatever convenient method is required, so long as time is advancing between calls to the procedure.

Whether using `CoSimTrans` or `CosimIrq`, the associated user program, `VUserMain`*`N`* need only register an interrupt callback function and then `SLEEPFOREVER` (a macro provided via the `OsvvmTrans` class header). The callback function, when activated,

can then update interrupt state, and make it available to other user programs and/or threads. A simplified code snippet of this arrangement is shown below.

```
static uint32_t irq        = 0;
static int      irqBarrier = 0;

// Event processing
extern "C" void VUserMain0()
{
    OsvvmCosim mgr(0);

    // Event loop
    while (true)
    {
        waitOnIrqBarrier();
        processIrq(intReq, mgr);
    }
}

// Interrupt callback function
int procIrq(int int_vec)
{
    irq = int_vec;
    irqBarrier++;
}

// Interrupt node program
extern "C" void VUserMain1()
{
    OsvvmCosim int(1);

    int.regInterruptCb(procIrq);

    SLEEPFOREVER;
}
```

In this example the call to `processIrq` hides a lot of detail. At its simplest it would make calls to methods that generate transactions to service the interrupts, but it might also be within a multi-threaded implementation, as discussed earlier, posting event messages to different threads to handle the specific classes of events and be the source of the generation of transactions. This architecture would make the event loop non-blocking and so will log all incoming events, including edge triggered interrupts of a single cycle. The software can then set a pending bit for the interrupt, which is cleared when it gets serviced. The interrupt node program might

take care of this, providing means to clear the pending state, but a better way could be in a software model of an interrupt controller if co-simulating, for example, within a software model of an SoC system, with the controller having configurable registers to mark interrupt relevant inputs as edge triggered and thus register, and hold, pending status internally.

With this kind of arrangement, the interrupt state within the software will be updated regardless of whatever activity is happening on any transaction interface—even within the middle of an active transaction.

### Modelling an Event Based Software Architecture

Using the separated interrupt handing node, an event based software architecture can be modelled that has an event loop. The interrupt node software is constructed as described above to simply receive the changes in interrupt state and make it available. It can also use a barrier, as described earlier, which it updates on changes to the interrupt state, to allow synchronisation from external functions.

A user software program associated with a transaction generating node, such as a `CoSimTrans` based loop for address bus model independent transaction generation, can then be constructed where the code is a loop waiting on the barrier from the interrupt node, in a manner we looked at earlier. When released, the code can then process the new state to call various functions to perform the required actions for the current interrupt state, which will be, ultimately, the transaction calls to handle the interrupt. This main program loop, then, is the event loop servicing the incoming events on the interrupt lines.

## Conclusions

In this article we had a look at the use of multiple nodes to drive more than one source of model independent transactions, looking at how to add and use this to the OSVVM VHDL co-simulation environment, and what this means in terms of user code.

We saw that the user code for each node run as separate threads, but that the co-simulation layer co-ordinates this in such a manner that data exchange is safe between the user programs, without further thread requirements on that program, and that even synchronisation between the user code is possible with emulating a simple barrier variable.

We also looked at how the co-simulation environment can even allow a node's user program to be multi-threaded, if so required, and accesses to transaction generation from multiple threads, via the APIs, is thread safe. Becoming more advanced, we looked at driving multiple nodes from a single user program, with its own multiple threads. We finished up by looking at having a separate interrupt node, and how this might be used to model an event loop. Thus, the OSVVM co-simulation environment provides the facilities for any level of user code sophistication, from simply writing linear, single threaded, test code to drive a DUT, to co-simulating with multi-threaded, event based, embedded software models.