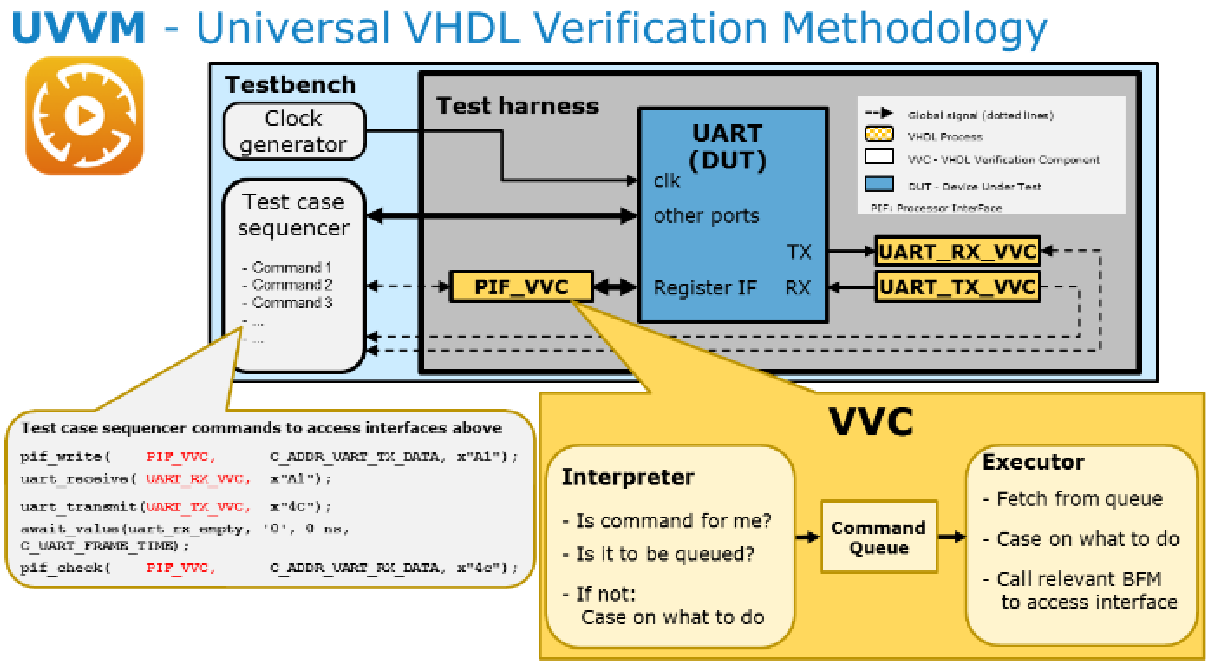

```
pif_write(    PIF_VVC,      C_ADDR_UART_TX_DATA, x"A1");
uart_receive( UART_RX_VVC,  x"A1");
uart_transmit(UART_TX_VVC,  x"4C");
await_value(uart_rx_empty, '0', 0 ns,
C_UART_FRAME_TIME);
pif_check(    PIF_VVC,      C_ADDR_UART_RX_DATA, x"4c");
```

Advanced VHDL Verification
- Made simple - For anyone

Part 3: The VHDL Verification Component (VVC)

In part 1 (*The testbench architecture*), you could see that any HW/FPGA designer could easily understand, modify, extend, maintain and reuse a UVVM based testbench architecture, - resulting in a huge efficiency improvement. Then in part 2 (*The testbench sequencer*) you could see that the commands to control the input stimuli and output checking were just like simple software test sequencers, where you control everything and easily understand what is happening.
So now you might wonder 'Where has all the complexity gone? Has it all ended up in chaotic verification components?'
- But no, - the verification components have a very structured micro architecture that is almost the same for all Verification Components, making it easy to make new VVCs from an automatically generated template. So let's have a closer look at this.

Simple Verification Component architecture

As you can see from the figure above, the VVC has a straight forward, easily understandable architecture.

It consists of three main parts: A Command Interpreter, a Command Queue and a Command Executor.

Interpreter

The Interpreter checks the commands from the central test sequencer to see if a given command is targeted at this particular VVC (e.g. the PIF_VVC). If so - then it checks whether the command is to be put on the Command Queue for further handling by the Executor - or to be handled immediately by the Interpreter itself. If the latter - then the interpreter checks the command type and immediately performs the requested operation. This could typically be to flush the queue or fetch a previous result, but it could also be to wait for a specific (or all) queued command to finish - via an 'await_completion()' command.
An important part of receiving a command from the central test sequencer is a handshake where the interpreter - by acknowledging a command - allows the central test sequencer to continue with the next command. This handshake is non time consuming, meaning that the sequencer may distribute lots of commands to one or many VVCs at the same instant in time.
The only time this handshake may be time consuming and thus stalling the sequencer, is for an 'await_completion()' command when the command(s) to wait for is not yet completed. This means the 'await_completion()' command is great for synchronization and yields a good overview of the sequence of events inside the testbench.

Queue

The Command Queue is like the name indicates, just a standard queue for commands to be forwarded to the Executor. Standard procedures like 'put', 'get' and 'flush' are used to access and control the queue.

Executor

The Command Executor executes commands contiguously as long as there are commands available in the queue. The Executor checks the command type and performs the requested operation. This could typically be to write or read&check a register in a VVC for PIF, AXI-lite or Avalon MM - or to transmit or receive&check data in a VVC for UART, SPI or I2C. It could of course also be to execute more advanced protocols like Ethernet or any proprietary protocol. In fact - the executor - or more correctly some of the executor commands - are often the only differences between two separate VVCs - like a UART and SPI. This means the only major difference between some VVCs may be the actual BFM procedures - like uart_transmit() versus spi_transmit().

Going from BFM procedures to VVCs

I think all FPGA designers today agree that BFM procedures are needed for any reasonably acceptable testbench. I also think most FPGA designers do make BFM procedures as needed for their FPGA and module testbenches. This is great and definitely sufficient for simple verification scenarios. However, for slightly more complex verification scenarios, VVCs are much better. This applies:
- When you need to stimulate or check multiple interfaces simultaneously
- When you want to have additional concurrent checks - e.g. bit rate margins
- When you have a pipelined or out of order protocol
- When you need a more structured architecture or better overview
- etc....

A major benefit of UVVM is that the overhead of going from a BFM to a VVC is close to nothing compared to the development time of the BFM it self. As a matter of fact you can for instance make a UART VVC in less than 30 minutes provided you have already made the BFM procedures.

VVC similarities and generation

As all VVCs share the same or much of the same architecture, it is easy to understand and implement new VVCs. UVVM even comes with a Python script that generates a template for new VVCs for you. Currently there are free and opens source VVCs available for UART, AXI4-lite, simple Avalon MM, I2C and SBI (simple bus interface). It is the intention that users make their own VVCs for their own proprietary protocols, and hopefully we will also soon see more open source VVCs from various contributors.

Key benefits

The beauty of UVVM is that major modifications and extensions are possible within the given framework while preserving the structure. This is the overview, modifiability, extendibility, maintainability and reuse you get with UVVM VVC Framework.

Structure and overview - throughout

In part 1 you have seen that the VVC Framework testbench architecture is easily understandable by anyone. In part 2 you saw that the commands to control the input stimuli and output checking are just like simple software test sequencers, where you control everything and understand what is happening. And now you have seen the simple structure and the reusability of the VVCs.

So - in summary - you have now seen how UVVM is all about structure, overview, readability, extendibility, maintainability and reuse.
Now all you have to do is to try it out in your next project. I'm pretty sure you will very soon see the huge benefits of this system and methodology.

UVVM is free and open source, and you can use it for anything you like, with no restrictions other than the standard MIT open source license.
UVVM is available from github.com and bitvis.no (released in Feb 2016).

A two-day course was held in Stockholm April 20-21 on how to make good and structured VHDL testbenches. UVVM was used as an example on how to do this. Most probably there will also be a course in Germany in September (date and place TBD).