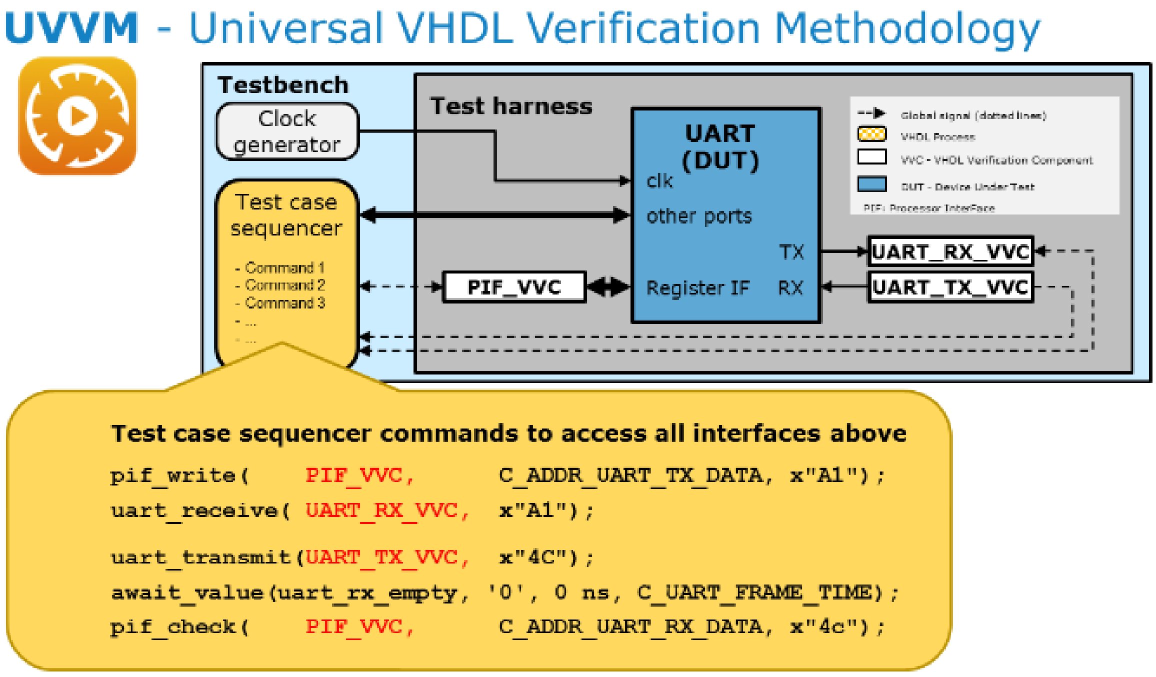


```
pif_write(     PIF_VVC,       C_ADDR_UART_TX_DATA, x"A1");
uart_receive( UART_RX_VVC,  x"A1");
uart_transmit(UART_TX_VVC,  x"4C");
await_value(uart_rx_empty, '0', 0 ns, C_UART_FRAME_TIME);
pif_check(     PIF_VVC,       C_ADDR_UART_RX_DATA, x"4c");
```

Advanced VHDL Verification
- Made simple - For anyone

Part 2: The testbench sequencer

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. The remaining main issue is then 'How do we control the stimuli and check the outputs of the DUT (Device Under Test) in such a system?

Simple and understandable test sequencer commands

The simplest answer to this question is to show an excerpt of the test sequencer for the testbench in part 1. In the figure above you can see how the DUT access can be handled as follows:
1. Tell PIF_VVC (Processor IF VVC) to write x"A1" to the UART TX register
2. Tell UART_RX_VVC to expect to receive that data (x"A1") from UART TX
3. Tell UART_TX_VVC to transmit x"4C" into UART RX.
4. Wait until the signal 'uart_rx_empty' (part of 'other ports') is deactivated - indicating that UART RX has received this data (with a time-out after C_UART_FRAME_TIME)
5. Then tell PIF_VVC to read the UART RX register and check that this is x"4C".

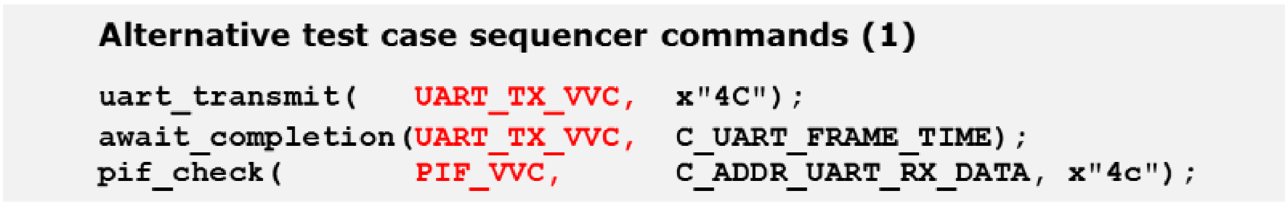
These commands look like pseudo code and can be matched directly against your verification/test specification. This allows you to control everything in your simple to understand testbench architecture - with simple to understand software like commands. Nothing happens by magic, and you are in total control -with full overview of what your testbench is doing. Note that UART_TX_VVC is a model for a DUT-external transmitter, and thus connected to the UART receiver.

Just a distribution of BFM procedure execution

Analysing the commands a bit closer you see that they look like ordinary BFM (Bus Functional Model) procedures, and in fact the only thing that differentiates these commands from ordinary BFMs is that they are not executed by the test sequencer, but distributed to the verification components and executed there. The distribution itself doesn't take time, which means that the test sequencer can distribute multiple command in series at the same time. In the example above commands 1,2,3 are distributed and started simultaneously, and thus we suddenly have a very structured methodology for controlling and checking multiple interfaces in parallel.

Simple synchronization of transactions

The 'await_value()' command will make the test sequencer wait for the requested signal change (uart_rx_empty going to '0') , and thus synchronize further events. Another way of synchronizing events is to use the 'await_completion()' command as shown below. This command tells UART_TX_VVC to stall the test sequencer until UART_TX_VVC has finished all its pending commands.



Alternative test case sequencer commands (1)

```
uart_transmit(   UART_TX_VVC,  x"4C");
await_completion(UART_TX_VVC,  C_UART_FRAME_TIME);
pif_check(       PIF_VVC,      C_ADDR_UART_RX_DATA, x"4c");
```

As the verification components all have their internal command queue, the sequencer may in fact distribute multiple commands to the same verification component in zero time - to be executed by the verification component in sequence - back to back.

Easy control of parallel activity - to reach cycle related corner cases

The VVC Framework also allows you to easily skew the BFM execution time on multiple interfaces with respect to each other. In the example below, after 'await_completion()' is finished, two commands ('insert_delay ()'and 'pif_check()') are both distributed to PIF_VVC to be executed in sequence there. This means there will be a clock period delay until 'pif_check()' is executed. Then of course you could have small command sequences within a loop - or a higher level procedure - with a parameter controlling the skew and data. So far, all examples have shown direct control from the central test sequencer in the testbench. This is a good start, but the VVC Framework allows more advanced features for smarter control to be included in a simple way.

Alternative test case sequencer commands (2)

```
uart_transmit(   UART_TX_VVC,   x"4C");
await_completion(UART_TX_VVC,   C_UART_FRAME_TIME);
insert_delay(    PIF_VVC,       1 * _CLK_PERIOD);
pif_check(       PIF_VVC,       C_ADDR_UART_RX_DATA, x"4c");
```

Advanced verification mechanisms - still easy

Example a) below shows the 'uart_transmit()' command, where the data to be transmitted is replaced by the enumerated 'FROM_BUF' and constant 'C_BUF1' - indicating that data is to be fetched from buffer number C_BUF1, and that 256 bytes should be transmitted. The buffer may have been pre-filled by the sequencer. The same buffer will of course be used to check the transmitted data.
Example b) shows that UART_TX_VVC may also be asked to generate its own random data and put these into buffer number C_BUF2, and transmit 256 bytes.
Example c) takes example b) one step further by **not** telling UART_TX_VVC how many bytes to transmit, but rather telling it to continue transmission until a predefined data coverage is reached.

```
a) uart_transmit(UART_TX_VVC,   FROM_BUF, C_BUF1, 256);
```

```
b) uart_transmit(UART_TX_VVC,   RANDOM_TO_BUF, C_BUF2, 256);
```

```
c) uart_transmit(UART_TX_VVC,   RANDOM_TO_BUF, C_BUF1, DATA_COVERAGE);
```

These examples all show how more intelligence can be moved from the test sequencer to the verification components and further improve overview, readability, extendibility, maintainability and reuse. What UVVM provides here is a structured architecture, a structured way of transferring test sequencer commands to the verification components, and structured verification components that can handle extensions in a structured way. So in case you haven't got the message - UVVM is all about structure and simplicity :-)
And as we all know from the design side, a bad architecture results in more complex code and numerous iterations, - whereas a good architecture results in a far faster implementation and better quality.

UVVM users have full freedom to make anything they want

UVVM users may make their own verification components and make new command variants as shown in examples a,b,c. The users also choose the target names, the command names, the command parameters and how the commands are executed in the verification components.

Key benefits

The beauty of UVVM is that major modifications and extensions are possible within the given framework while preserving the structure and the overview. This is the overview, modifiability, extendibility, maintainability and reuse you get with UVVM VVC Framework, - a free and open source Universal VHDL Verification Methodology (UVVM), - available from github.com and bitvis.no (released only a few weeks ago).

Structure and overview - throughout

In part 1 you have seen that the VVC Framework testbench architecture is easily understandable by anyone, - and now you have seen 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.
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. I'll come back to this is part
3. https://www.linkedin.com/pulse/advanced-vhdl-verification-made-simple-3-espen-tallaksen