Mentor®
A Siemens Business

1100010110100111101001110110110100111100 11

UVVM
Advanced VHDL verification – made simple

Webinar, May 20, 2020

(by Espen Tallaksen)

What is UVVM?

UVVM = Universal VHDL Verification Methodology

- Open Source Verification Library & Methodology

- Very structured infrastructure and architecture
- Significantly improves Verification Efficiency
- Assures a far better Design Quality

- Extremely fast adoption by the world-wide VHDL community

- Recommended by Doulos for Testbench architecture
- Supported by more and more EDA vendors
- ESA projects to extend the functionality

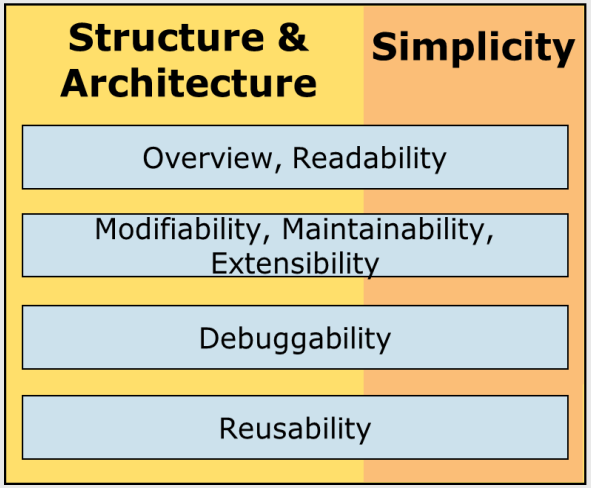
DOULOS

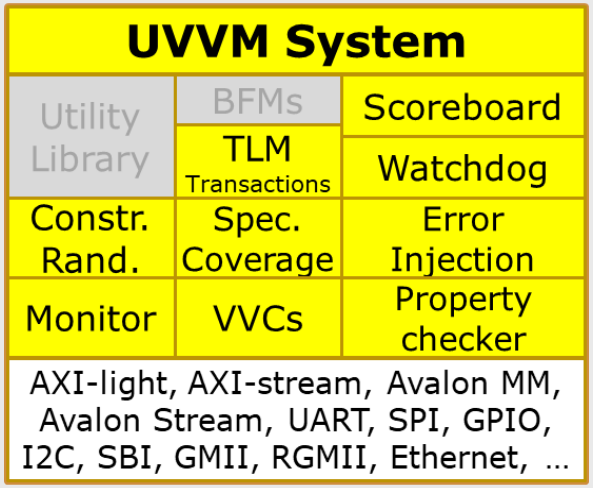
esa

UVVM enables Quality and Efficiency

UVVM System handles this for low to high complexity DUTs

UVVM Utility Library handles this for low complexity DUTs





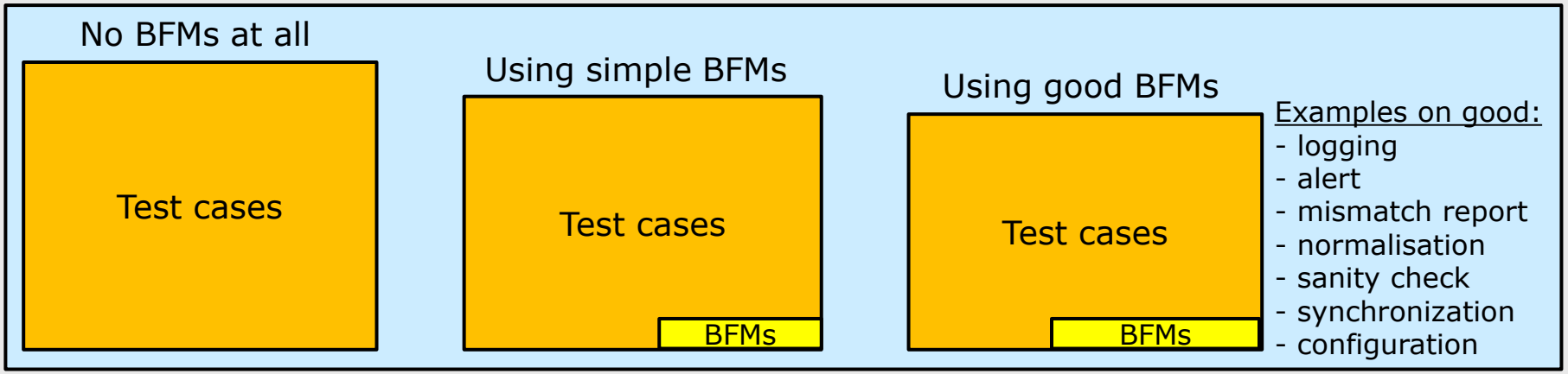
Huge improvement potential in most projects	
Save **100-1000 hours** in low-medium complexity projects Save **500-3000 hours** in medium to high complexity projects	**+ TTM** **+ MTBF** **+ LCC**

bitvis

Overview, Readability, Maintainability, Ext….

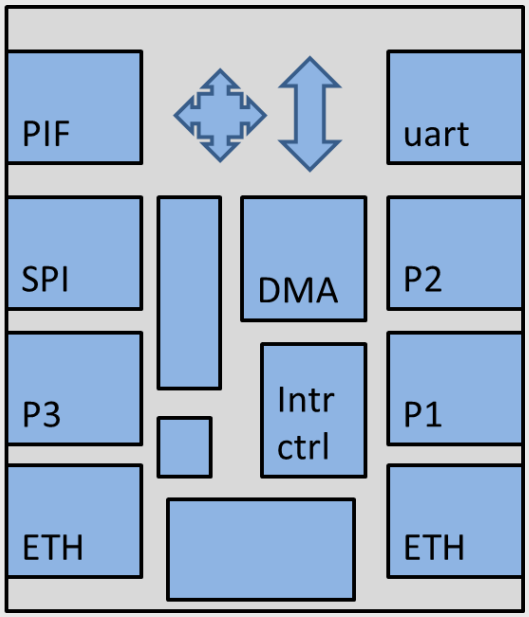
- Cannot get this for free – with no added complexity
  - A **Design** has modules, bus system, clock system, reset system, …
  - Some designers prefer not to make procedures, functions, variables, …
  - BFMs **should** contain more than just the signal wiggling/sampling
  - ➔ **Must** prioritise complexities for different parts of any system

Example: **Total** FPGA verification workload
for a **simple** testbench using different BFM strategies:



bitvis

More complex design challenges



New challenges:

➔ handle any number of interfaces in a structured way?

➔ reuse major TB elements between module TBs?

➔ reuse major module TB elements in the FPGA TB?

Extended challenges:

➔ read the test sequencer almost as simple pseudo code?

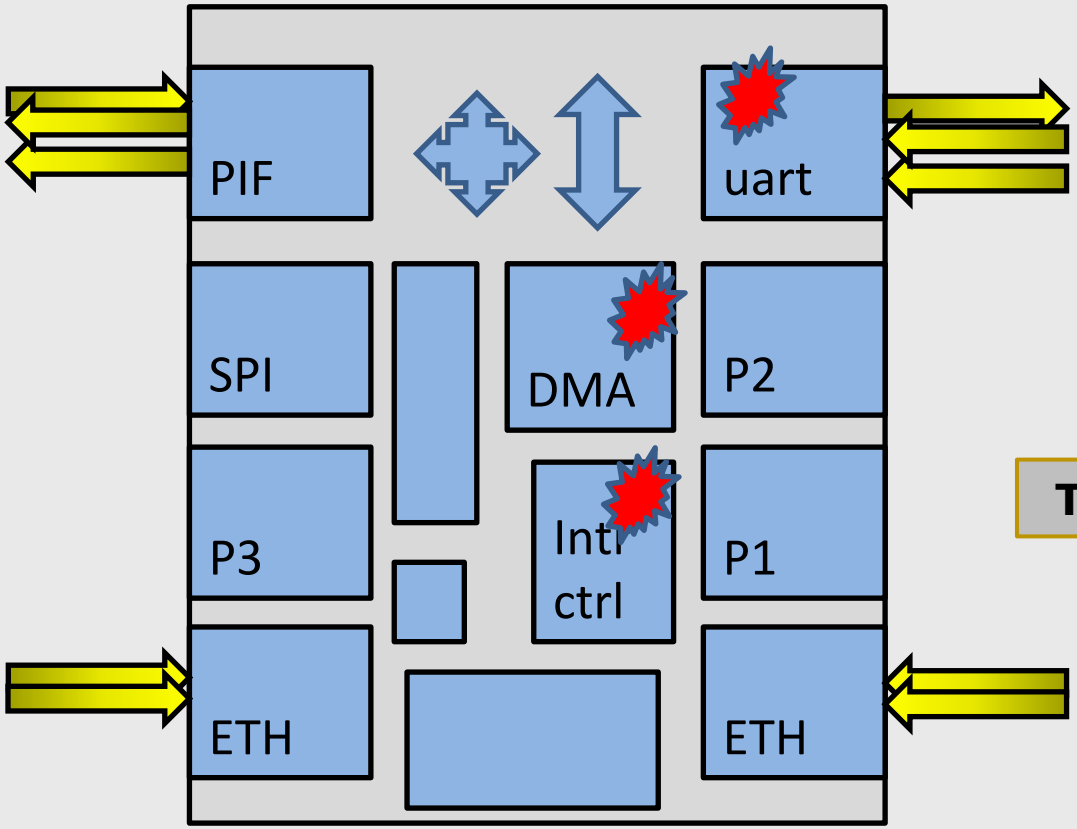
➔ recognise the verification spec. in the test sequencer?

➔ understand the sequence of event
- just from looking at a single test sequencer

➔ reach cycle related corner cases

bitvis

More complex design challenges



Sequential testing

➔ Find some bugs?

➔ Then behaves fine

➔ Value related corners?

This is a significant problem!

Parallel operation

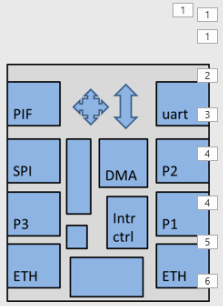
➔ Multi-end cycle related corners

➔ Bugs are hard to find

bitvis

Need a structured approach

More complex design challenges



New challenges:

➔ handle any number of interfaces in a structured way?

➔ reuse major TB elements between module TBs?

➔ reuse major module TB elements in the FPGA TB?

Extended challenges:

➔ read the test sequencer almost as simple pseudo code?

➔ recognise the verification spec. in the test sequencer?

➔ understand the sequence of event
- just from looking at a single test sequencer

➔ reach cycle related corner cases

Solution:

- A component oriented TB
- Known functionality throughout
- Commands that even SW designers can understand
  - And even **Write**

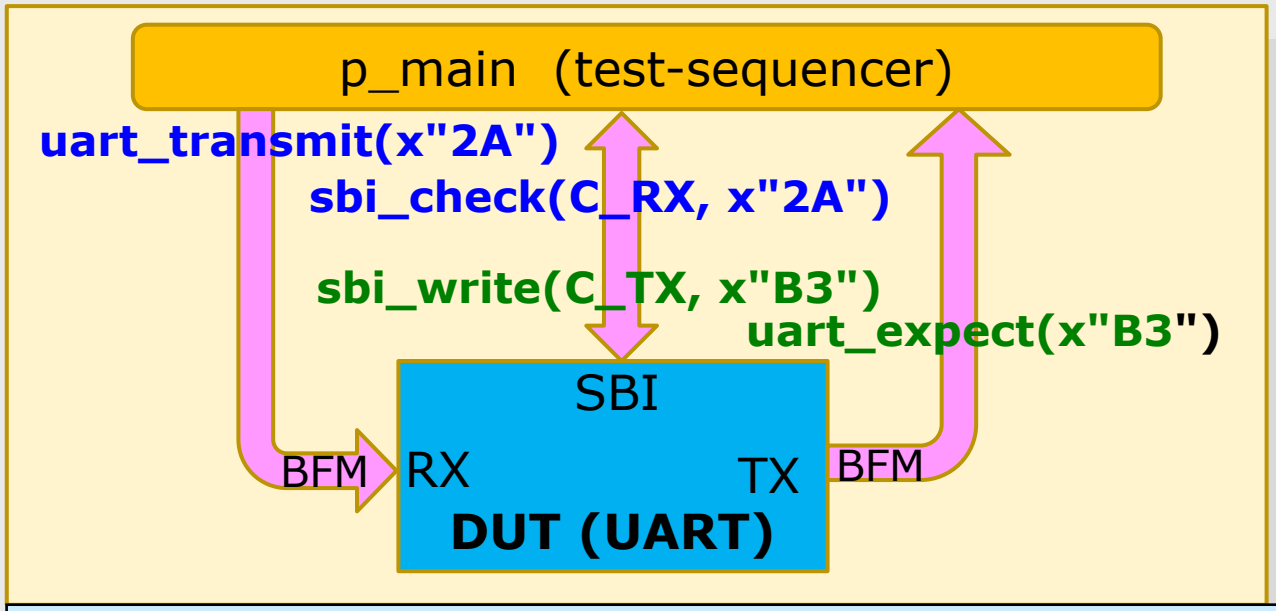
- As for BFMs
  - Hide protocol and details
  - Increase readability
    - where it matters the most
  - Allow some BFM internal complexity
    - to save time where it matters the most
- Priority 1 – for efficiency

  Reduce time for writing test cases

➔ Requires:
  - Simple test case commands
  - Easy understandable TB architecture
  - No need to look into interface handlers
  - All synchronization via test sequencer
    - Preferably only one

bitvis

Simple data communication



Free, Open source BFMs:

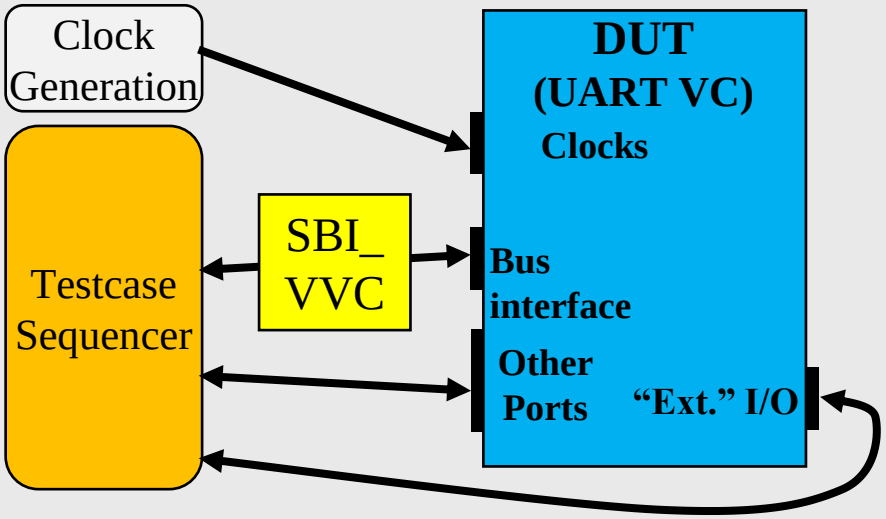
UART, AXI4-lite, SPI, I2C, Avalon MM, AXI4-stream, Avalon stream, GPIO, SBI, GMII, RGMII, ...

```
TB:  172 ns. uart_tb    uart_transmit(x2A) on UART RX
TB:  192 ns. uart_tb    sbi_check(x1, ==> x2A) completed. From UART RX
```

```
TB:  192 ns. uart_tb    sbi_write(x2, ==> xB3) completed. To UART TX
TB: ERROR:
TB:     192 ns. uart_tb
TB:             value was: 'xB2'.  expected 'xB3'.
TB:             (From uart_expect(xB3))
TB:=============================================================
```


bitvis

VVC – In its simplest form



Going from BFM to VVC

Using Bus access (SBI) as example
- E.g. write to a register in DUT

Sequencer command using BFM:

```
sbi_write(C_ADDR_TX, x"2A");
```

Minimum VVC

1. Interpret command from sequencer **in zero time**
2. Execute respective BFM towards DUT

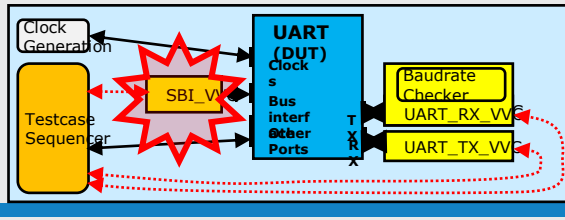
Sequencer command using VVC:

```
sbi_write(SBI_VVCT,1, C_ADDR_TX, x"2A");
```

→ Results in above BFM being executed from VVC towards DUT

Very simple VVC – already allows simultaneous execution of BFMs on different interfaces

Verification Component



SBI_VVC

Interpreter

- Is command for me?
- Is it to be queued?
- If not:
  Case on what to do

Command Queue

Executor

- Fetch from queue
- Case on what to do
- Call relevant BFM(s)
  & Execute transaction

Same main architecture in every VVC

- >95% same code in Interpreters
- Same command queue
- 95% same code in Executors - apart from BFM calls

VVC Generation

UART BFM to UART_VVC:
less than 30 min

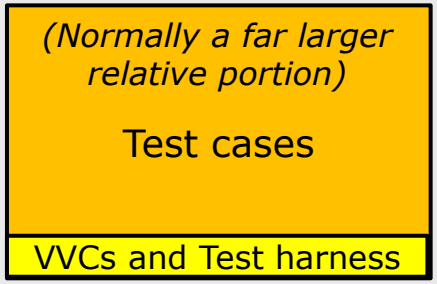
bitvis

Controlling your test case

- Writing and debugging test cases is by far the most time consuming verification task

  ➔ **Highest priority for efficiency improvement**

- **Must support this…**



Simple, understandable transaction commands

```
axilite_write(), axilite_check()
axistream_transmit(), axistream_expect()
```

Standardized:
- command distribution syntax
- handling of multiple instances
- transfer of commands to VVCs

Synchronization of all interfaces from one single file

```
await_completion()
insert_delay()
```

Standardized:
- synchronization of VVC
- alignment of interface actions

Critical for efficiency, quality & reuse

Common commands for VVC control – out of the box

```
terminate_current_command()
fetch_result ()
```

Standardized:
- common VVC commands
- Multicast and Broadcast

Automatically available

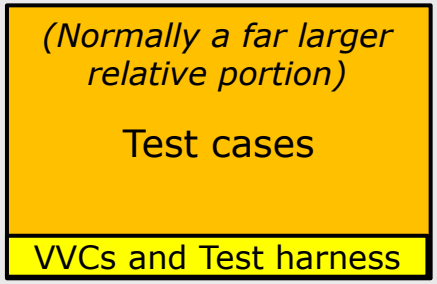
bitvis

Controlling your test case

- Writing and debugging test cases is by far the most time consuming verification task

  ➔ **Highest priority for efficiency improvement**

- **Must support this…**




```
transmit(AXISTREAM_VVCT,1, v_packet1, ...);
transmit(AXISTREAM_VVCT,2, v_packet2, ...);
transmit(AXISTREAM_VVCT,3, v_packet3, ...);
...sequencer doing something else
```

Simplified test case section

Starting activity simultaneously to check for contention or cycle related bugs

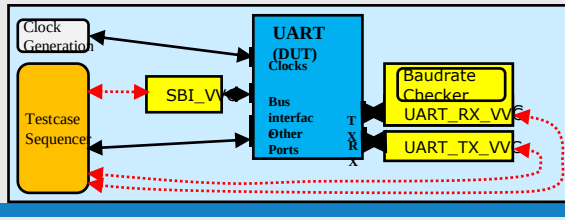
```
transmit(AXISTREAM_VVCT,1, v_packet1, ...);
transmit(AXISTREAM_VVCT,2, v_packet2, ...);
insert_delay(AXISTREAM_VVCT,3, C_CLK_PERIOD, ...);
transmit(AXISTREAM_VVCT,3, v_packet3, ...);
...sequencer doing something else
```

Simplified test case section

Skewing interface activity to check for contention or cycle related bugs

bitvis

VVCs: Extended



- **Easy to handle split transactions**
- **Easy to add local sequencers**
- **Easy to add checkers/monitors/etc**
- **Easy to handle out of order execution**

Interpreter

- Is command for me?
- Is it to be queued?
- If not:
  Case on what to do

***_VVC**

Command Queue

Executor

- Fetch from queue
- Case on what to do
- Call relevant BFM(s)
  & Execute transaction

Queue

Response-Executor

DUT

Scoreboard

Transaction Info

CR + FC

Bit-rate checker

Frame-rate checker

Gap checker

bitvis

All VVCs have their own Quick Reference

AXI4-Stream VVC – Quick Reference

For general information see UVVM VVC Framwork Essential Mechanisms located in uvvm_vvc_framework/doc. **CAUTION**: shaded code/description is preliminary

AXI4-Stream Master

In order to use the AXI4-Stream VVC in master mode, it must be instantiated in the test harness by setting the generic constant *'GC_MASTER_MODE'* to TRUE.

axistream_transmit[_bytes] (VVCT, vvc_instance_idx, data_array, [user_array, [strb_array, id_array, dest_array]], msg, [scope])

Example: axistream_transmit(AXISTREAM_VVCT, 0, v_data_array(0 to v_numBytes-1), v_user_array(0 to v_numWords-1), " Send a 'v_numBytes' byte packet to DUT");
axistream_transmit(AXISTERAM_VVCT, 0, v_data_array(0 to v_numBytes-1)(31 downto 0), v_user_array(0 to v_numWords-1), "Send a '4 x v_numBytes' byte packet to DUT");

Note! Use axistream_transmit_bytes ( ) when using t_byte_arr

bfm_sync	t_bfm_sync	SYNC_ON_CLOCK_ONLY
byte_endianness	t_byte_endianness	FIRST_BYTE_LEFT
valid_low_at_word_num	integer	0
valid_low_duration	integer	0
check_packet_length	boolean	false
protocol_error_severity	t_alert_level	ERROR
ready_low_at_word_num	integer	0
ready_low_duration	integer	0
ready_default_value	std_logic	'0'

1 VVC procedure details

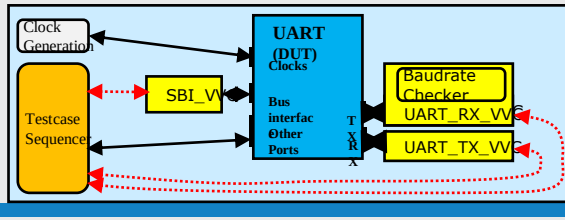
Procedure	Description
axistream_transmit[_bytes]()	axistream_transmit[

The axistream_transm
commands have comp
the AXI4-Stream BFM
The axistream_transm
'GC_MASTER_MODE' to true.

Examples:

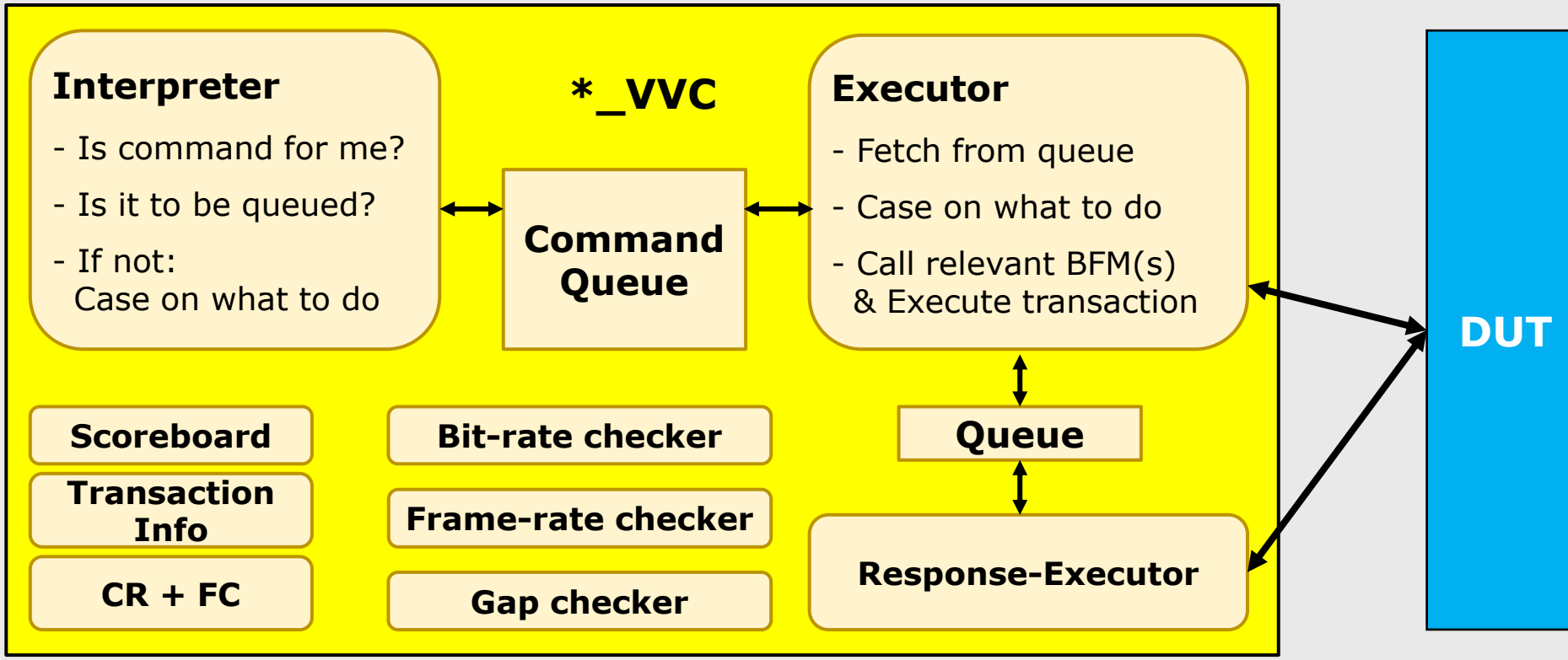
```
axistream_transmit(AXISTREAM_VVCT, 0, v_data_array(0 to 1),
                   "Send a 2 byte packet to DUT, tuser=0 each word / clock cycle", C_SCOPE);
axistream_transmit(AXISTREAM_VVCT, 0, v_data_array(0 to 1)(16 downto 0),
                   "Send a 4 byte packet to DUT, tuser=0 each word / clock cycle", C_SCOPE);
axistream_transmit(AXISTREAM_VVCT, 0, v_data_array(0 to v_numBytes-1), v_user_array(0 to v_numWords-1),
                   " Send a 'v_numBytes' byte packet to DUT", C_SCOPE);
axistream_transmit(AXISTREAM_VVCT, 0, v_data_array(0 to v_numBytes-1), v_user_array(0 to v_numWords-1),
                   v_strb_array(0 to v_numWords-1), v_id_array(0 to  v_numWords-1),
                   v_id_array(0 to v_numWords-1), "Send..", C_SCOPE);
axistream_transmit(AXISTREAM_VVCT, 0, (x"D0", x"D1", x"D2", x"D3"), (x"00", x"0A"),
                   "Send a 4 byte packet with tuser=A at the 2nd (last) word", C_SCOPE); --(tdata'length = 16)
axistream_transmit(AXISTREAM_VVCT, 0, (x"D0", x"D1", x"D2", x"D3"), (x"00", x"00", x"00", x"0A"),
                   "Send a 4 byte packet with tuser=A at the 4th (last) word", C_SCOPE); --(tdata'length = 8)
```

VVCs: Too advanced???



Advanced functionality is great when needed, but what if not???

- If using an existing VVC, just ignore it. Use it out of the box – without the extras.
- If making your own VVC, don't include the advanced stuff

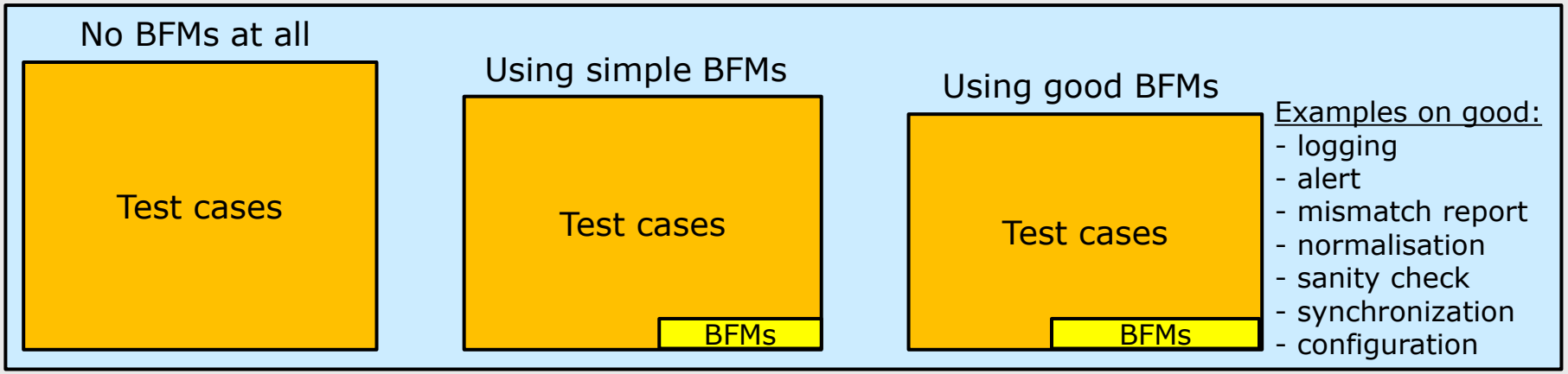


bitvis

Overview, Readability, Maintainability, Ext.... - Revisited

- Cannot get this for free – with no added complexity
  - A **Design** has modules, bus system, clock system, reset system, …
  - Some designers prefer not to make procedures, functions, variables, …
  - BFMs **should** contain more than just the signal wiggling/sampling
  - ➔ **Must** prioritise complexities for different parts

Example: **Total** FPGA verification workload
for a **simple** testbench using different BFM strategies:



bitvis

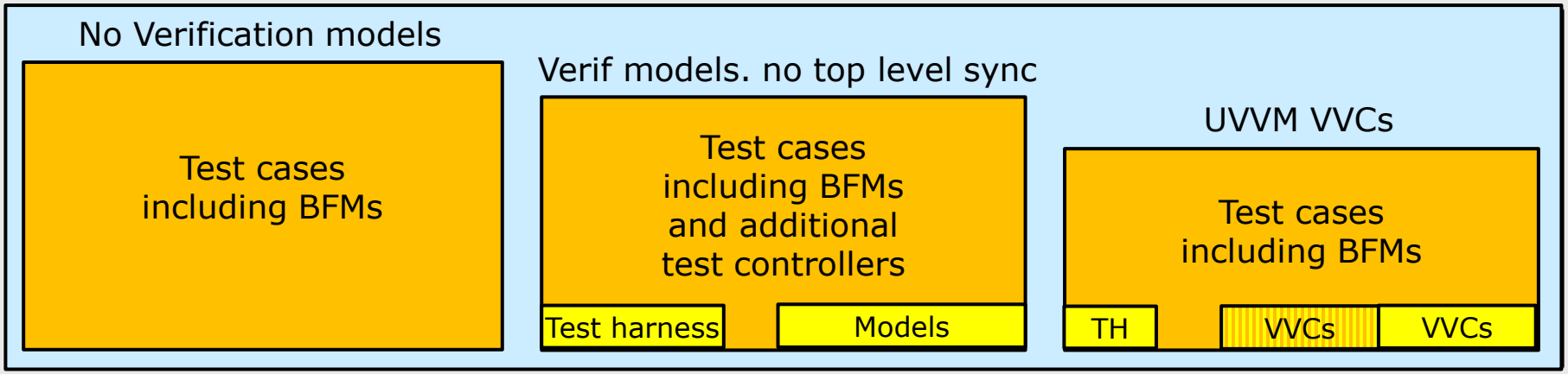
Overview, Readability, Maintainability, Ext.... - **For more complex Systems or Modules**

- Having some of these challenges

  → Need to prioritise simplicity
- Could have just encapsulated BFMs

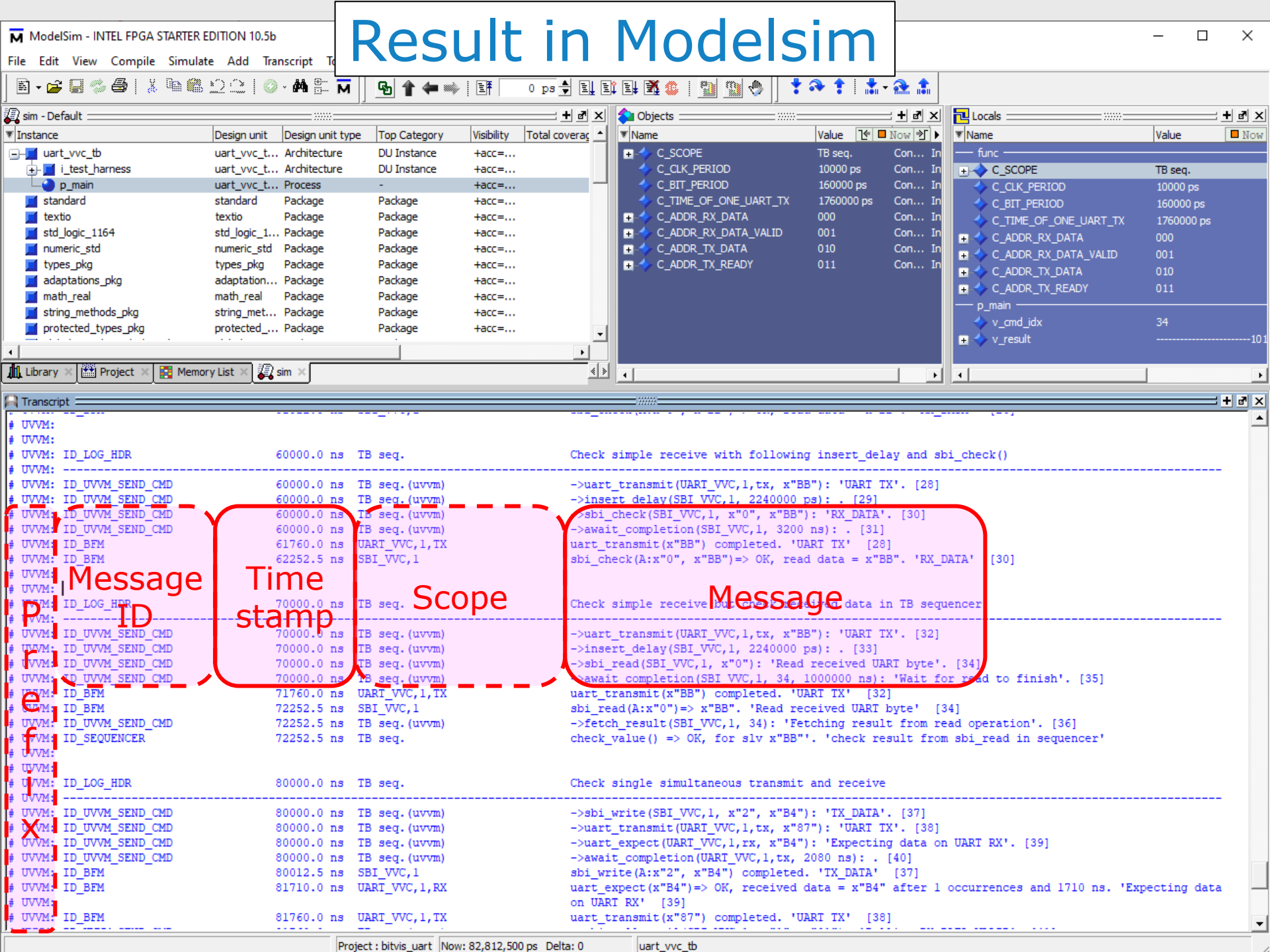
  → Local reuse fine
  → Very complex synchronization from sequencer
- Could have local sequencers with BFMs and sync. via semaphores/flags

  → Difficult to reuse these when semaphores are system dependent
  → Still really complex synchronization



bitvis

Result in Modelsim



Prefix

Message ID

Time stamp

Scope

Message

Example on transcript/log

Show all commands

```
60000.0 ns  TB seq.          Check uart_transmit() with following insert_delay and sbi_check()
-------------------------------------------------------------------------------------------
60000.0 ns  TB seq.(uvvm)    ->uart_transmit(UART_VVC,1,tx, x"BB"): 'UART TX'. [28]
60000.0 ns  TB seq.(uvvm)    ->insert_delay(SBI_VVC,1, 2240000 ps): . [29]
60000.0 ns  TB seq.(uvvm)    ->sbi_check(SBI_VVC,1, x"0", x"BB"): 'RX_DATA'. [30]
60000.0 ns  TB seq.(uvvm)    ->await_completion(SBI_VVC,1, 3200 ns): . [31]
61760.0 ns  UART_VVC,1,TX    uart_transmit(x"BB") completed. 'UART TX'  [28]
62252.5 ns  SBI_VVC,1        sbi_check(A:x"0", x"BB")=> OK, read data = x"BB". 'RX_DATA'  [30]
```

Show BFM execution of commands only

```
60000.0 ns  TB seq.          Check uart_transmit()with following insert_delay and sbi_check()
-------------------------------------------------------------------------------------------
60000.0 ns  TB seq.(uvvm)    ->uart_transmit(UART_VVC,1,tx, x"BB"): 'UART TX'. [28]
60000.0 ns  TB seq.(uvvm)    ->insert_delay(SBI_VVC,1, 2240000 ps): . [29]
60000.0 ns  TB seq.(uvvm)    ->sbi_check(SBI_VVC,1, x"0", x"BB"): 'RX_DATA'. [30]
60000.0 ns  TB seq.(uvvm)    ->await_completion(SBI_VVC,1, 3200 ns): . [31]
61760.0 ns  UART_VVC,1,TX    uart_transmit(x"BB") completed. 'UART TX'  [28]
62252.5 ns  SBI_VVC,1        sbi_check(A:x"0", x"BB")=> OK, read data = x"BB". 'RX_DATA'  [30]
```

Example on transcript/log

Show distribution of commands only

```
60000.0 ns  TB seq.          Check uart_transmit()with following insert_delay and sbi_check()
-------------------------------------------------------------------------------------------
60000.0 ns  TB seq.(uvvm)    ->uart_transmit(UART_VVC,1,tx, x"BB"): 'UART TX'. [28]
60000.0 ns  TB seq.(uvvm)    ->insert_delay(SBI_VVC,1, 2240000 ps): . [29]
60000.0 ns  TB seq.(uvvm)    ->sbi_check(SBI_VVC,1, x"0", x"BB"): 'RX_DATA'. [30]
60000.0 ns  TB seq.(uvvm)    ->await_completion(SBI_VVC,1, 3200 ns): . [31]
61760.0 ns  UART_VVC,1,TX    uart_transmit(x"BB") completed. 'UART TX'  [28]
62252.5 ns  SBI_VVC,1        sbi_check(A:x"0", x"BB")=> OK, read data = x"BB". 'RX_DATA'  [30]
```

Show headers only

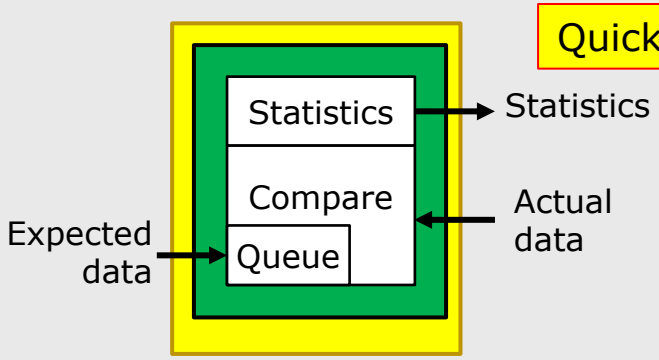
```
60000.0 ns  TB seq.          Check uart_transmit()with following insert_delay and sbi_check()
-------------------------------------------------------------------------------------------
60000.0 ns  TB seq.(uvvm)    ->uart_transmit(UART_VVC,1,tx, x"BB"): 'UART TX'. [28]
60000.0 ns  TB seq.(uvvm)    ->insert_delay(SBI_VVC,1, 2240000 ps): . [29]
60000.0 ns  TB seq.(uvvm)    ->sbi_check(SBI_VVC,1, x"0", x"BB"): 'RX_DATA'. [30]
60000.0 ns  TB seq.(uvvm)    ->await_completion(SBI_VVC,1, 3200 ns): . [31]
61760.0 ns  UART_VVC,1,TX    uart_transmit(x"BB") completed. 'UART TX'  [28]
62252.5 ns  SBI_VVC,1        sbi_check(A:x"0", x"BB")=> OK, read data = x"BB". 'RX_DATA'  [30]
```

Example on transcript/log

Assumed most normal: View execution towards interfaces only

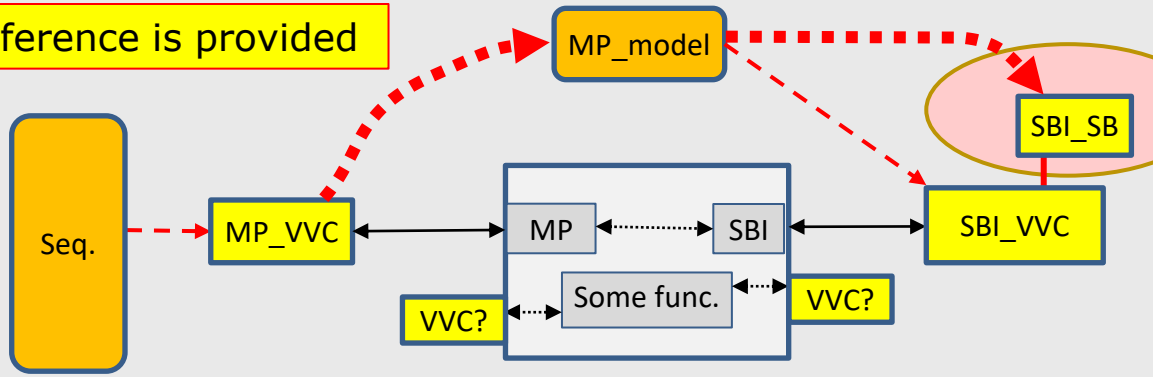
```
60000.0 ns  TB seq.          Check uart_transmit()with following insert_delay and sbi_check()
-------------------------------------------------------------------------------------------
61760.0 ns  UART_VVC,1,TX    uart_transmit(x"BB") completed. 'UART TX'  [28]
62252.5 ns  SBI_VVC,1        sbi_check(A:x"0", x"BB")=> OK, read data = x"BB". 'RX_DATA'  [30]
```

Generic Scoreboard



generic data type

Quick Reference is provided



- logging/reporting
- flushing queue
- clearing statistics

Configuration record:

- allow_lossy
- allow_out_of_order
- mismatch_alert_level
- etc...

Counting:

- entered
- pending
- matched
- mismatched
- dropped
- deleted
- initial garbage

- insert, delete, fetch
- ignore_initial_mismatch
- indexed on either entry or position
- optional source element (in addition to expected + actual)

add_expected()

add_expected ([instance], expected_element, [msg], [source_element])
add_expected ([instance], expected_element, tag_usage*, tag, [msg], [source_element])

***NOTE: tag_usage can only be TAG**

Inserts expected element at the end of scoreboard.

msg_id: ID_DATA

Example:

```
uart_sb.add_expected(x"AA");
uart_sb.add_expected(1, x"AA", TAG, "byte1", "Insert byte 1", x"A");
uart_sb.add_expected(1, x"AA", "Insert byte 1", x"A");
```

check_actual()

check_actual ([instance], actual_element, [msg])
check_actual ([instance], actual_element, tag_usage*, tag, [msg])

***NOTE: tag_usage can only be TAG**

Checks received data against oldest element in the scoreboard. If the element is found, it is removed from the scoreboard and the matched-counter is incremented. If the element is not found, an alert is triggered and the mismatch-counter is incremented. If out-of-order is allowed the scoreboard is searched from front to back. The mismatch-counter is incremented if no match is found. If lossy is allowed the scoreboard is searched from front to back. If a match occurs, the match counter is incremented and the preceding entries dropped. If no match occurs, the mismatch counter is incremented.

msg_id: ID_DATA

Example:

```
uart_sb.check_actual(v_received_byte);
uart_sb.check_actual(2, v_received_byte, "Check byte1"); -- Check received data against scoreboard elements in instance 2
uart_sb.check_actual(3, v_received_byte, TAG, "byte1", "Check byte 1"); -- Check received data against scoreboard
                                                                        -- elements with the TAG "byte1" in instance 3
```

flush()

flush ([instance], [msg])

Deletes all entries in the specified instance. Can also be called with parameter ALL_INSTANCES.

msg_id: ID_DATA

Example:

```
uart_sb.flush(VOID);
uart_sb.flush(2);
uart_sb.flush(ALL_INSTANCES);
```

Alert and Report from Scoreboard

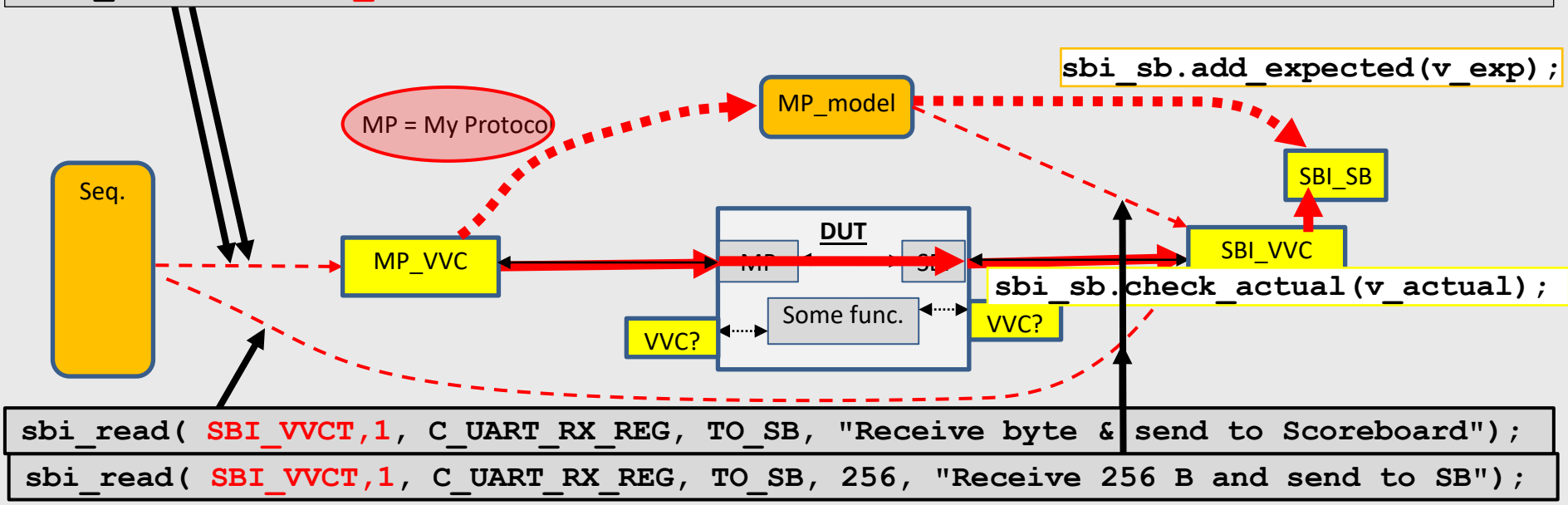
```
================================================================================
                        *** SCOREBOARD COUNTERS SUMMARY: UART Scoreboard ***
================================================================================
             ENTERED   PENDING   MATCH   MISMATCH  DROP  INITIAL_GARBAGE  DELETE  OVERDUE CHECK
instance: 1      4         3         1         0         0                0         0         0
instance: 2      7         5         2         0         0                0         0         0
================================================================================
```

```
# UVVM: ================================================================
# UVVM: ***  ERROR #1  ***
# UVVM:        9070 ns   UART Scoreboard
# UVVM:                  check_actual() instance 1 => MISMATCH, expected: x"0C"; actual: x"FF".
# UVVM:
# UVVM: Simulator has been paused as requested after 1 ERROR
```

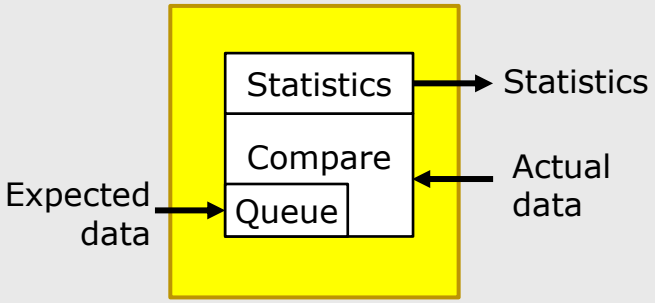
```
================================================================================
                        *** SCOREBOARD COUNTERS SUMMARY: UART Scoreboard ***
================================================================================
             ENTERED   PENDING   MATCH   MISMATCH  DROP  INITIAL_GARBAGE  DELETE  OVERDUE_CHECK
instance: 1      4         0         3         1         0                0         0         0
instance: 2      7         0         7         0         0                0         0         0
================================================================================
```

Advanced verification
- Using Scoreboards, VVCs and Models

```
uart_transmit(UART_VVCT,1,TX,  x"42", "Transmit single byte");
uart_transmit(UART_VVCT,1,TX,  256, RANDOM, "Transmit 256 rand bytes");
```

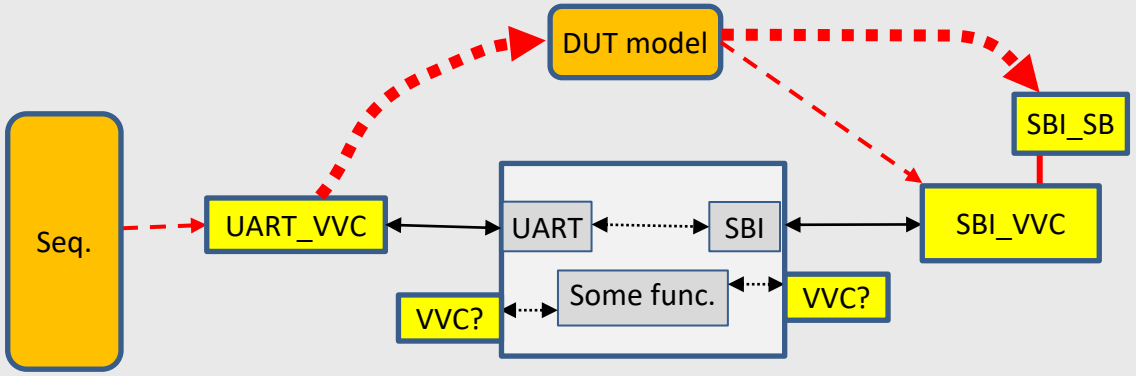


- **Model**
  - Models DUT behaviour
  - Receives transaction on DUT input
  - Generates expected data
  - Passes expected data onto Scoreboard



Advanced built-in funct.

- Ctrl randomisation and functional coverage
- Protocol aware Error injection
- Local sequencers
- Ctrl property checkers



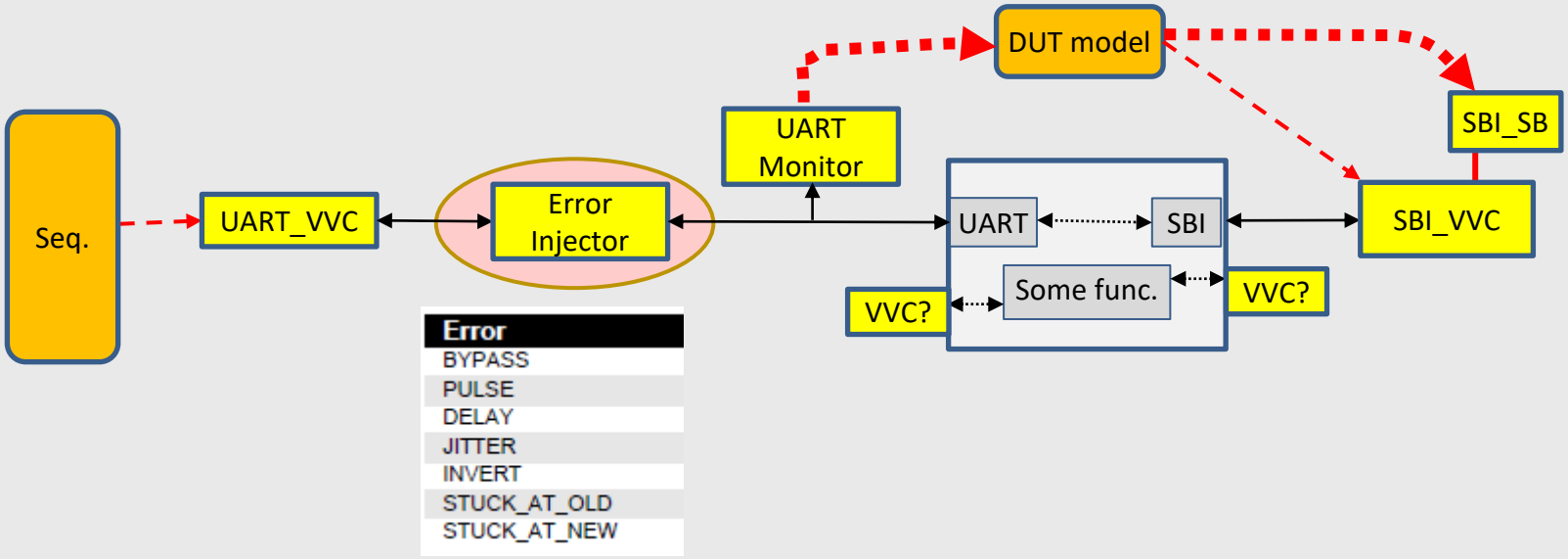
Randomisation Inside BFMs and VVCs Ex. UART/SBI

```
uart_transmit(UART_VVCT, 1, TX, x"AF", "Sending data to Peripheral 1");
uart_transmit(UART_VVCT, 1, TX, 5, RANDOM, "Sending 5 random bytes");
```

VVC Error injection record (inside the VVC configuration record above)

Record element	Type	DEFAULT	Description
parity_bit_error_prob	real	0,0	The probability that the VVC v
stop_bit_error_prob	real	0,0	The probability that the VVC v

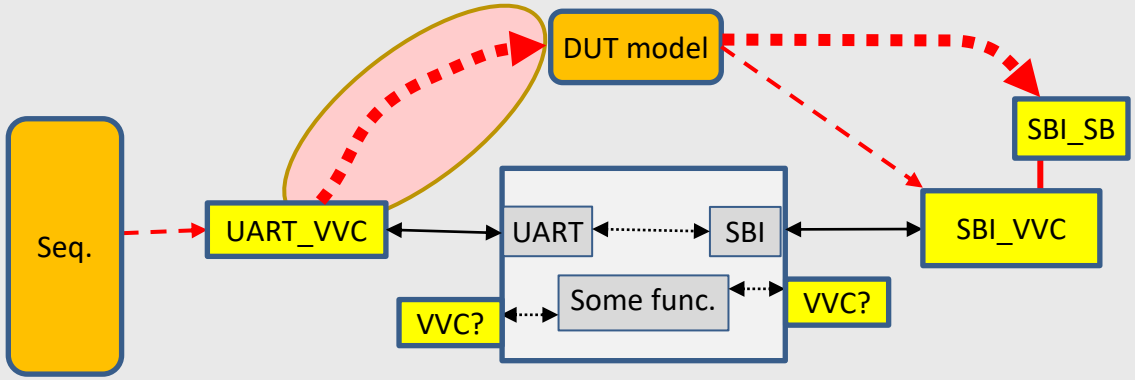
"Brute force" Error & Monitor



"Brute force" Error Injector	Dedicated VIP

Monitor	Module inside VIP	Ex. UART

Transaction info transfer

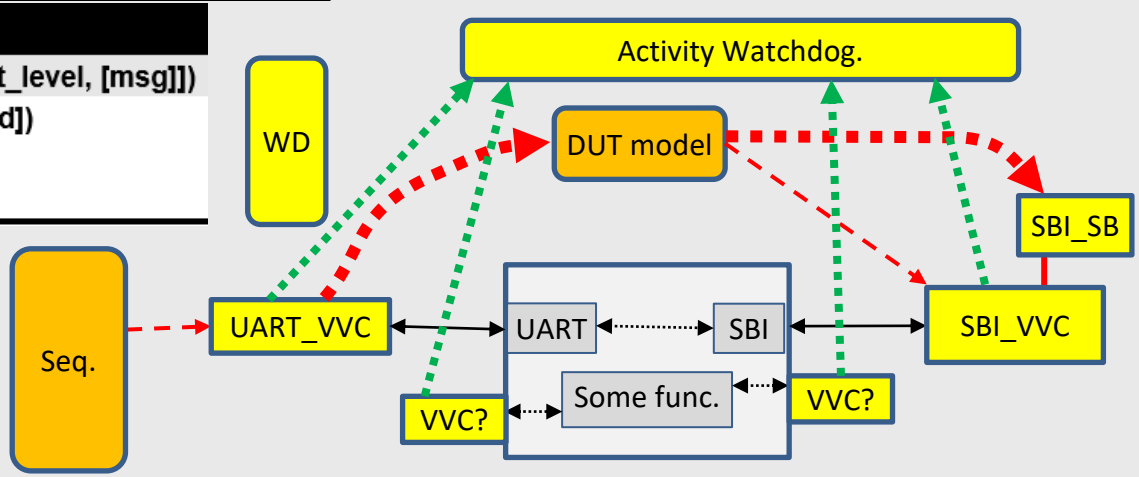


Transaction info	Inside VVC – Provided as a global signal

Watchdogs

Simple WD Inside Util

Watchdog
watchdog_timer(watchdog_ctrl, timeout, [alert_level, [msg]])
extend_watchdog(watchdog_ctrl, [time_extend])
reinitialize_watchdog(watchdog_ctrl, timeout)
terminate_watchdog(watchdog_ctrl)

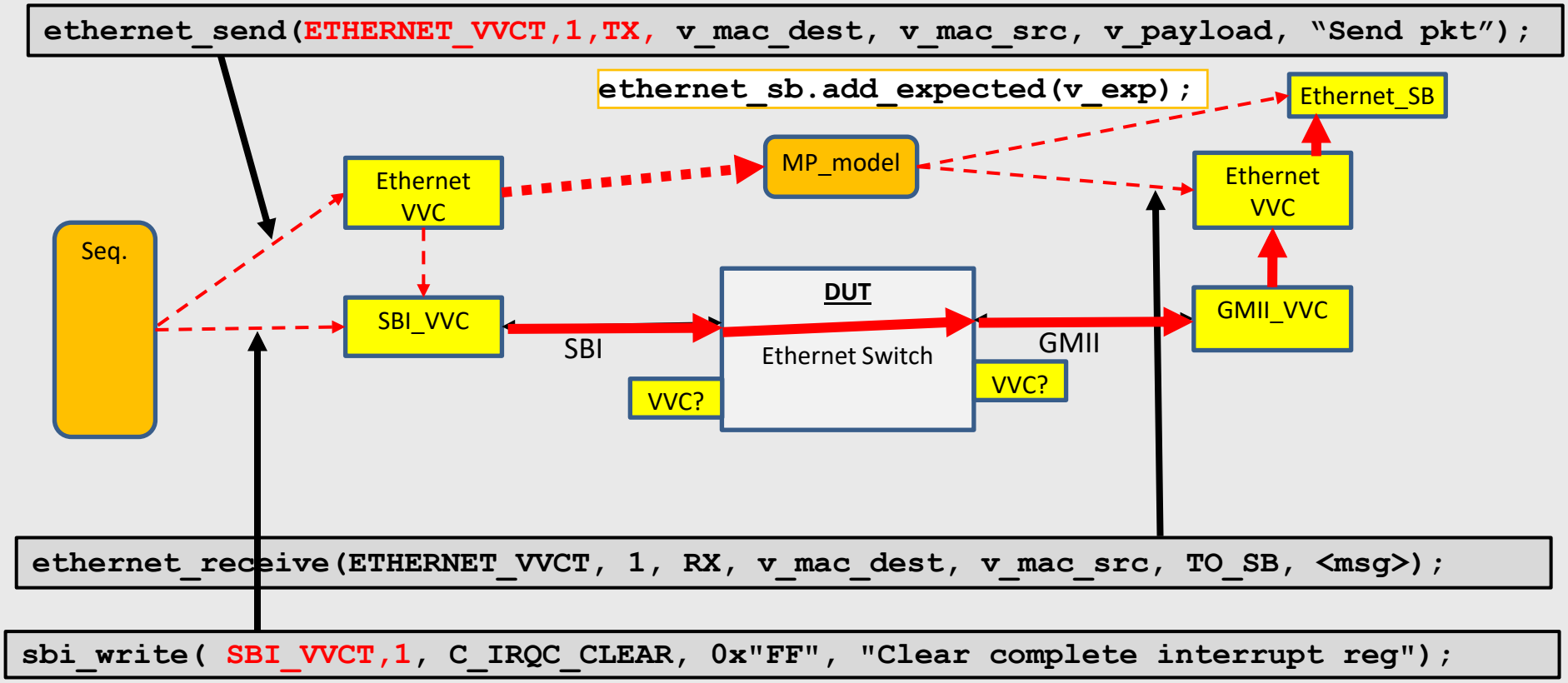


Apply both concurrently

Activity WD VVCs and UVVM

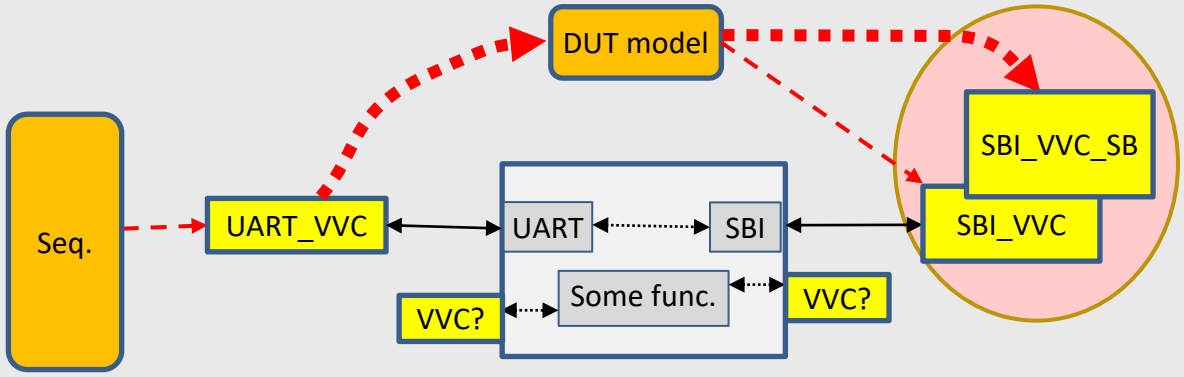
```
activity_watchdog(timeout, num_exp_vvc);
```

Hierarchical VVCs and SBs



Built-in Scoreboards

- Introduced for all pure SLV oriented VVCs
- Can then use TO_SB as a parameter in the VVC command

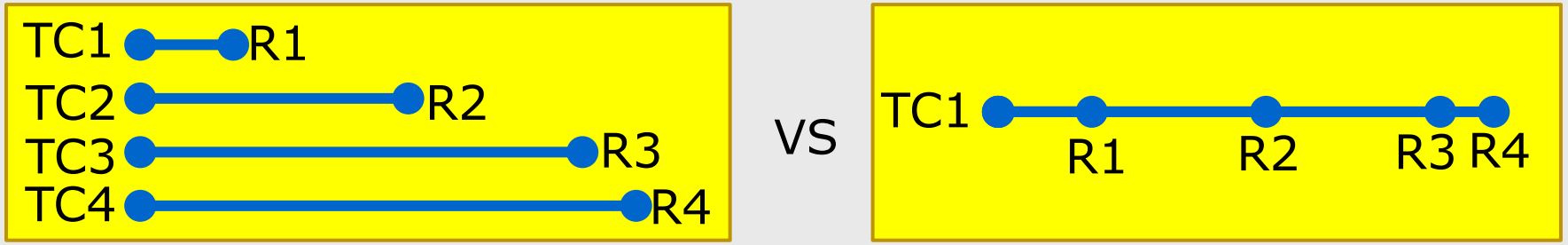



```
sbi_read( SBI_VVCT,1, C_UART_RX_REG, TO_SB, "Receive byte & send to Scoreboard");
```

Specification Coverage

- Assure that all requirements have been verified
  1. Specify all requirements
  2. Report coverage from test sequencer (or other TB parts)
  3. Generate summary report

- Solutions exist to report that a testcase finished successfully
  - BUT - reporting that a testcase has finished is not sufficient
- What if multiple requirements are covered by the same testcase?
  - E.g. Moving/turning something to a to a given position
    R1: Acceleration R2: Speed R3: Deceleration 4: Position etc..



Example case

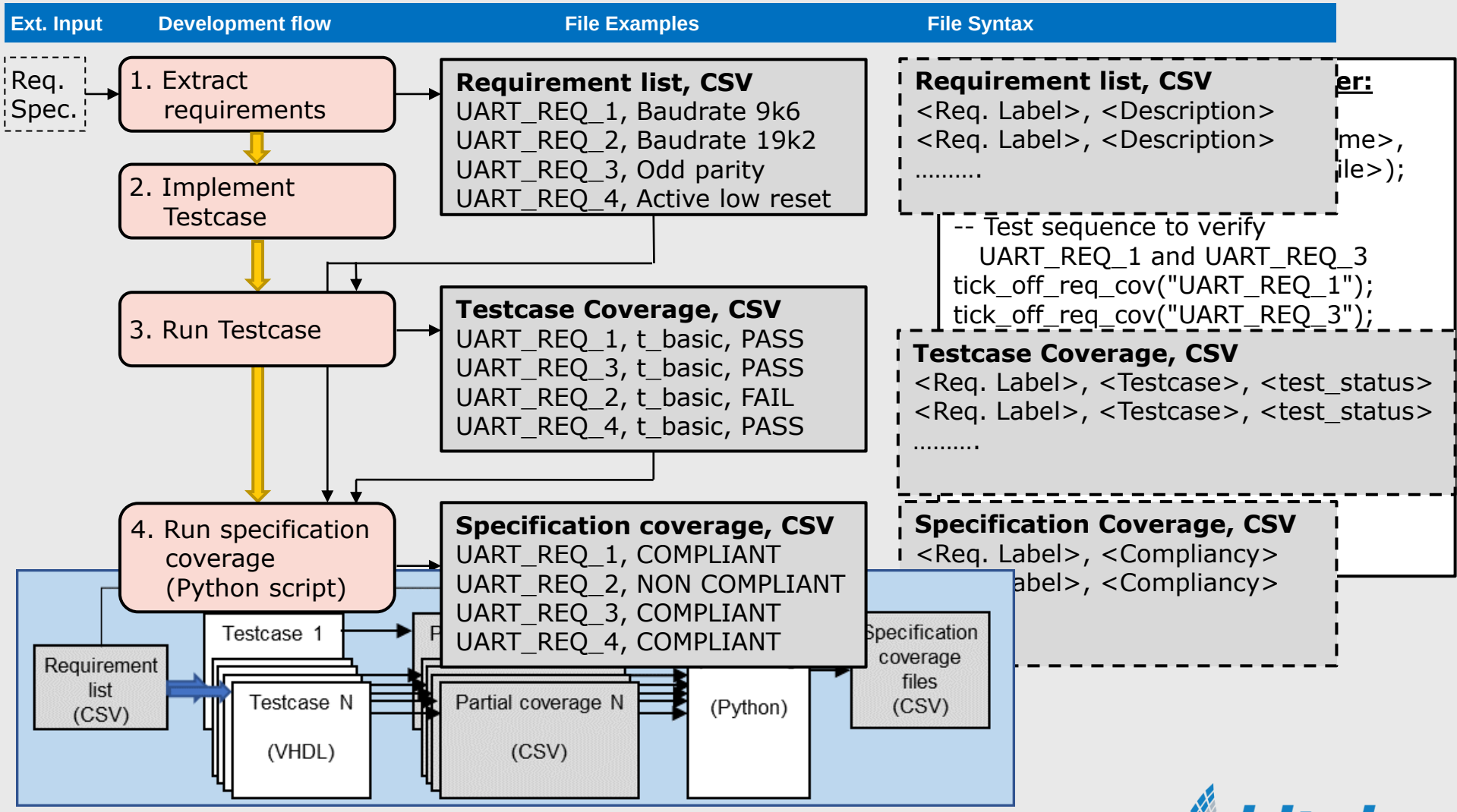
- UART
  - Showing only 4 requirements for simplicity

Requirement Label	Description
UART_REQ_1	The device UART interface shall accept a baud rate of 9600kbps.
UART_REQ_2	The device UART interface shall accept a baud rate of 19k2 bps.
UART_REQ_3	The device UART interface shall accept an odd parity
UART_REQ_4	The device reset shall be active low.

- Starting with a single testcase testing all requirements

Introduction & Simple Case

Simplified overview



bitvis

Advanced spec. cov.

Multiple reports:
E.g. Specification vs Verification matrices.

- May specify required testcase(s) for any given requirement
- May specify that at least one of multiple testcases must pass
- May specify that a requirement is tested in multiple testcases
- May have multiple requirement files
- May map requirements in one file to requirements in another
  - Thus allowing reusable TBs with coverage included
  - Also allows compound requirement to be split into multiple more detailed requirements

Levels of Freedom with UVVM

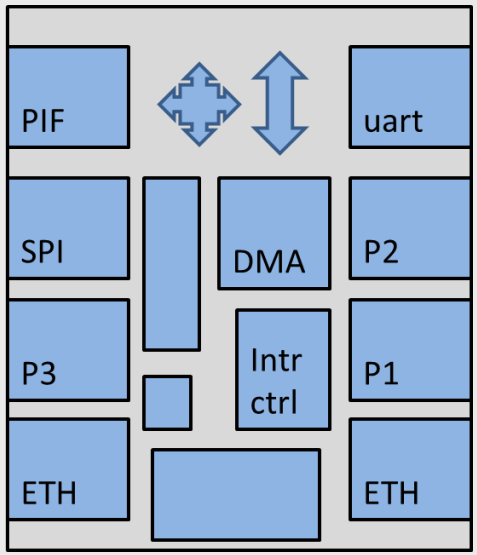
- You may use Utility Library for all or just parts of your TB
- You may use complete UVVM for all or just parts of your TB

- Non UVVM BFM or VVC may be wrapped to UVVM
- You may use from one to any number of UVVM BFMs/VVCs
  - and combine with any other approach – incl old legacy TBs

- You may combine with any other VHDL TB or VIP

- UVVM gives you a very well structured methodology
  - You can choose which parts you want to use

More complex design challenges - recap



New challenges:

- ➔ handle any number of interfaces in a structured way? ☑
- ➔ reuse major TB elements between module TBs? ☑
- ➔ reuse major module TB elements in the FPGA TB? ☑

Extended challenges:

- ➔ read the test sequencer almost as simple pseudo code? ☑
- ➔ recognise the verification spec. in the test sequencer? ☑
- ➔ understand the sequence of event
  - just from looking at a single test sequencer ☑
- ➔ reach cycle related corner cases ☑

UVVM – Usage still exploding

Recommended by Doulos for Testbench Architecture

ESA project is extending the UVVM functionality

If **including** VHDL, Verilog, SystemVerilog, System-C, etc:

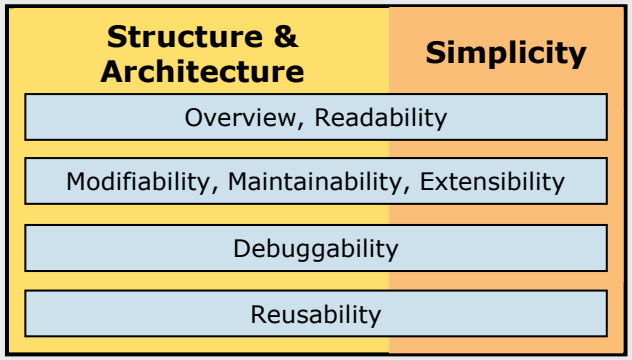
- UVVM usage from <1% to 26% in Europe in only 2 years…
  → Increased a lot since then.

Fastest growing verification methodology in the VHDL strongholds.
(much faster than UVM/SystemVerilog over the last two years – all languages included)

*1: According to Wilson Research, October 10, 2018 (Survey executed spring 2018)

Summary

- Huge improvement potential for far faster FPGA verification
- UVVM is a game changer for efficiency and quality



UVVM is Open Source

UVVM may save 200-3000 hours on medium to high complexity project

github.com/UVVM

forum.uvvm.org

uvvm.org

Also see my LinkedIn profile for a number of articles on UVVM++

bitvis

Mentor®
A Siemens Business

Espen Tallaksen