

## Use Scripting Language in Combination with the Verification Testbench to Define Powerful Test Cases. *by Franz Pammer, Infineon*

This article focuses on providing a methodology to make the process of writing test cases easier for verifying mixed analog digital designs. It explains the advantage of using a scripting language together with a Hardware Description Language (HDL) environment.

It shows a way to give the HDL engineers the possibility to use constrained randomized test cases without having too much knowledge about the implemented testbench.

Two possible scenarios for defining your test cases will be demonstrated. One can be used for the traditional VHDL testbench and another for the SystemVerilog (SV) Open Verification Methodology (OVM) environment. For both cases a scripting language will be used to write test cases.

At the end you will understand that with the mentioned methodology also design engineers will fully benefit of verifying their design.

Please note that within this article Tool command language (TCL) is used as scripting language. The reason for using TCL is, that Questasim is controlled by TCL and also has some more built in TCL commands included. Nevertheless this approach can be used with any scripts which are supported by your simulation environment.

### WAYS OF MAKING VERIFICATION EASIER FOR HDL ENGINEERS

Nowadays designers are experienced in optimizing the circuits for power, area and implementing the algorithm effectively. Moreover, they can write very good VHDL testbenches with advanced stimulus generation and file I/O parsers, but they do not have the deep knowledge to develop a constrained randomized configurable testbench environment to verify all corner cases of the design.

Besides the designers you will find the verification engineers who have deep knowledge of modern software design and applying the newest techniques on today's higher verification languages like SV together with

the OVM libraries. They are able to generate constrained randomized stimuli's with automatic checkers, scoreboards, analyzing all kinds of coverage models.

In most of the cases it is nearly impossible to distinguish between design and verification engineers. In many projects design engineers will do a basic verification of their module on their own. Reasons can be lack of resources or that the design engineer just wants to make a short rough verification of his designed module by himself. In order to avoid any misunderstanding during specification or implementation of the design, the four eyes principle is recommended. Meaning a different engineer should verify the block generated by another design engineer.

It is important to emphasize that it is really strongly recommended to make the verification process itself easy, reusable and powerful as possible. In this way only a limited amount of verification experts are needed who will build up the testbench and provide a simple interface for defining test cases in all variations. Such interfaces have to be as simple as possible but also as strong as possible. Furthermore, it should still be feasible to vary as much as possible the parameters and use the randomizing feature to catch all the corner cases. One common approach is to use SV OVM language. This methodology already provides easy test case definition by using OVM test files.

"The ovm\_test class defines the test scenario for the testbench specified in the test. The test class enables configuration of the testbench and environment classes as well as utilities for command-line test selection. [...], if you require configuration customization, use the configuration override mechanism provided by the SystemVerilog OVM Class Library. You can provide user-defined sequences in a file or package, which is included or imported by the test class." (OVM SystemVerilog User Guide, 2009) P. 67/68

If the OVM environment is well structured you can generate all possible test cases within your test file by simply defining some of your parameters (e.g. Use the appropriate sequence, correct register settings). In addition the

modification of such test files can easily be done by the design engineer. Depending on the setup you can define pure randomized test cases in your test file or directed test cases to verify dedicated use cases.

Several limitations should be pointed out with this approach. First, for every new test case a compilation has to be done before. Second, reusing a test file for other topics like test engineering is nearly impossible. In addition, it may take some time for design engineers to understand the process of writing test cases by these test files.

With respect to the above mentioned barriers another approach was developed. The idea is to use a scripting language in combination with a testbench like SV OVM to verify the AISC.

## TCL VERIFICATION METHODOLOGY

TCL is used by many simulator tools (e.g. Questa, NC-Sim) for controlling the simulators behavior. Normally TCL is used in the simulators only to start/stop simulation and generate some logging messages. Since TCL is a very powerful language, you can generate whole test cases by using TCL scripts. To this end the testbench only needs to provide an interface to the TCL script to allow taking control of the whole simulation flow. The interface usually will be built up by the verification expert who has deep knowledge of the testbench itself. Later, only the TCL script will be implemented by the design engineers to develop the test cases.

This methodology has many advantages which were described here:

- Same TCL file can be used for both VHDL/Verilog and SV OVM environment.  
In cases where you have a VHDL/Verilog testbench and in parallel a SystemVerilog OVM testbench. You can use the same TCL files. You just have to take care of the TCL interface
- If file is kept simply it can also be used for CV (Component Verification) and TE (Test Engineering) group as input source.

- Recompile is not needed.

If your simulator supports TCL you can simply try out your test case line by line within your simulator. You do not need to restart your simulation and change your test case file and try it out again.

- Other tasks or scripts can be called during your test execution
- Make debugging easier.

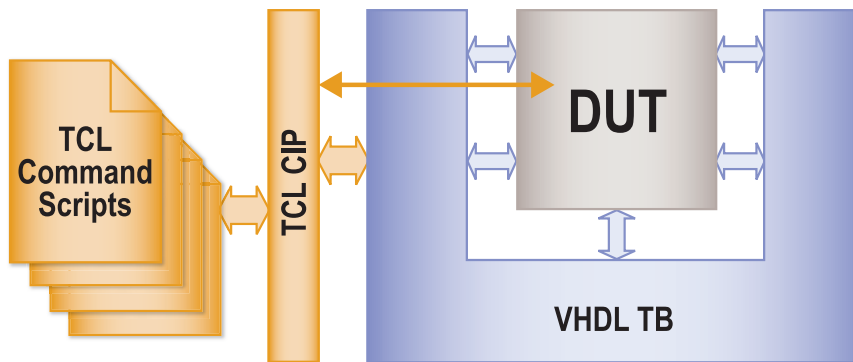
Simply think of analyzing some internal signals immediately or do some calculations to check for correctness. Temporary log signals to files during debug session.

## DESCRIPTION OF A TCL VHDL TESTBENCH

In traditional black box testing strategies the VHDL Testbench is surrounding the Design Under Test (DUT). The testbench controls all activities at the ports of the DUT. It generates the stimulus and checks all the responds from the DUT. The stimuli generation can be hard coded in some VHDL behavioral files, or defined in some ASCII text files. For reading in text files an extra file parser is needed which interprets the commands in the text file. The file parser will usually read in and execute the text file line-by-line. All the used commands need to be defined in the parser. For simple test cases this approach may fulfill all your needs. It can become complicated if you want to use some variables within your text file. You only have to let the files parsers know how to interpret variables. It will get more and more complicated if you want to do some math or use loops, branches and other conditions within your text file. Thus your VHDL parser will get more and more complex and future adoptions will be nearly impossible.

To overcome these disadvantages of using ASCII text file you can use simple TCL scripts. In scripting language you already have a huge functionality (e.g. handling of variables, loading another file, using if/while/loops). There is no need to take care of these items.

Figure 1 on the following page gives you an overview how such TCL scripts can be used within your environment.



**Legend:**



**Figure 1 : Overview TCL VHDL Testbench**

The TCL Command Interpreter (TCL-CIP) is the interface between the TCL command scripts and the Register Transfer Level (RTL) environment. It contains the libraries & packages for procedure/variable definitions, file I/O functionality etc. Basically it communicates via a well defined VHDL TB peripheral or it may also directly take control of some ports of the DUT or even some internal signals would be possible. The commands and variables which were defined in the CIP can be used in the TCL scripts.

Next is shown a short example how such an approach would look like. First is illustrated the TCL-CIP. For simplified reasons in this place there will not be packages and namespaces used, but also without using these techniques you can fully benefit with this methodology.

The TCL-CIP basically consists of three files.

1.) Global Signal definitions and variables

```
# Physical Environment Signals
set ENV_TEMPERATUR /analog_global_pack/
  temperature
set ENV_PRESSURE /analog_global_pack/pressure
set VBAT /analog_global_pack/vddd_global
```

```
# Clock signals
set CLK_SYS /tb/dcore/clock

# Clock period
set T_CLK_SYS "100ns"
```

**Listing 1 : TCL Global Signals**

2.) Tool specific procedure definition

As an example some procedures for the Questa simulator were shown. Whenever the tool version will be changed only this file needs to be modified.

```
#Source file of Questasim specific commands
proc run_proc { runtime } {
  run $runtime
}
# Set an internal signal, by using the force command
proc set_signal_proc {netname value } {
  force -freeze $signal $value 0
}
# Disable the force of an internal signal
proc noset_signal_proc {netname}{
  noforce $signal
}
```

**Listing 2 : TCL Questa Procedures**

### 3.) Low level procedure definition

Here all the above mentioned files were loaded together with the procedure definition which can be called from the TCL test cases.

```
# File: cip_procedures.tcl
# Load global signals
do $env (WORKAREA)/units/dcore/tcl/global_signals.tcl
do $env (WORKAREA)/bin/questa_procedures.tcl

set cmd_file $1.tcl; # This is the name of the
test case file
set log_file $1.log; # Logging information will be
written in that log file

proc log_test { logfileid logtxt }{
puts $logfileid $logtxt
echo $logtxt
}

# LOG: Just a comment which is written to the
transcript and to the file
proc LOG { args } {
global Now
global tb_g tb_g
set args [join $args]
set log "\# $args"
log_test $tb_g(logfileid) $log
}
# Simply delay the simulation by calling the
run command
proc DL {time args}{
...
run_proc $time
}

# Set a signal directly
proc SS { name args }{
global Now
global tb_g tb_g
set log "$Now : SS $name $args"
log_test $tb_g(logfileid) $log
upvar $name $name
set namex [set $name]
```

```
set log [set_signal $namex [lindex $args 0]]
}
# Cancele setting of a signal (NotSet)
proc NS { name args } {
...
set log "$Now : NS $name $args"
log_test $tb_g(logfileid) $log
set namex [set $name]
noset_signal $namex
}

set tb_g(logfileid) [open $logfileid w 0777]
# Check if command file exists
if {[file exists $cmd_file] == 0}{
puts stderr "Cannot open [exec pwd]/$cmd_file
exit
}
##### M A I N #####
do $cmd_file ; # Call the test case

puts $tb_g(logfileid) "TEST CASE RESULT : $TEST_
RESULT"
```

#### Listing 3 : TCL Basic Procedures

Below shows how such a TCL test case could look like:

```
# File: 001_BasicTest.tcl
LOG "Testcase: 000 Basic Test"
LOG "Author: MR.X"
SS MODE_A "01" ; # Set signal
DL 2500us ; # Delay for 2500us
LOG "Check power down signals"
for {set i 0} {$i < 10} {incr i 1} {
CL PD_ADC ; # Check low
CL PD_CPU
DL 100us
}
NS MODE_A ; # Cancel the Set command
LOG "Check power down signals"
# Wait until PD_MODE signal change to 1.
If this does not happen within 1ms then go on and #
generate an error
```

```
WT PD_MODE 1 1ms ; # Wait until PD_MODE
                    goes to '1' for 1ms
CH PD_ADC          ; # Check High
CH PD_CPU
LOG "Test finished"
```

**Listing 4 : TCL Test Case**

A startup file is needed which will execute your test case.

```
# File: startup.do
tcl_cip.tcl 001_BasicTest
```

**Listing 5 : Startup File**

The startup file is loaded with the vsim command.

```
vsim -c top_dut_tb-cfg.vhd -do startup.do
```

**Listing 6: Simulator start command**

## DESCRIPTION OF A TCL SV OVM TESTBENCH

The same approach as mentioned for the TCL-VHDL testbench can now be used also for the SV-OVM environment. Only for the TLC-CIP interface some adaptations are needed. Additionally within the OVM environment you need to write an extra TCL Interface OVC. This OVC reads via a virtual interface the commands from the TCL-CIP and takes control over other Open Verification Components (OVCs). Depending on the intention you can add a great deal of flexibility to these TCL scripts:

- Define the basic configuration
- Define which parameters to be randomized
- Define border values of randomized parameters
- Start dedicated sequencer
- Force or read back some internal signals
- Do some basic settings of the DUT
- Save Functional Coverage of your SystemVerilog Assertions (SVA)

If a SV OVM Testbench environment already exists new TCL functionality can easily be added even without having a TCL IF OVC. For the beginning it may be enough only to be able to force some internal signals or to do some interim calculation for other scripts.

Below you see an overview of how to connect your TCL script to the OVM environment. The TCL IF – OVC can be modeled in various ways. In our case it is built up like a usual OVC. The monitor is observing the TCL virtual interface for any activities. Whereas the driver will send some responds to this interface. The communication to the other OVCs can now be handled via ports or by using the sequencer as a virtual sequencer and in this way we can take control over all other sequencers within the OVM environment.

## CONCLUSION / SUMMARY

Using scripting languages like TCL can give some powerful features for writing test cases. By using a generic TCL interface these scripts can be used for various testbenches. No matter which HDL (Hardware Description Language) is used. Design engineers do not need to have much verification expertise for verifying their designs. Due to easier test case definition they also can do the basic verification of their design on its own. Additionally if these TCL scripts are structured well they also can be used for other topics too. Think of using these scripts as input for component verification to control the whole equipment. Furthermore you can reuse it for test engineering to configure the test setup. Therefore only the TCL-CIP needs to be adapted.

Depending on the available resources and timeframe the TCL functionality can be extended at any time. Extending your TCL functionality will not have too much impact on your existing testbench structure.

What I want to point out is that it is never too late to implement scripting functionality to your test case by using that approach.

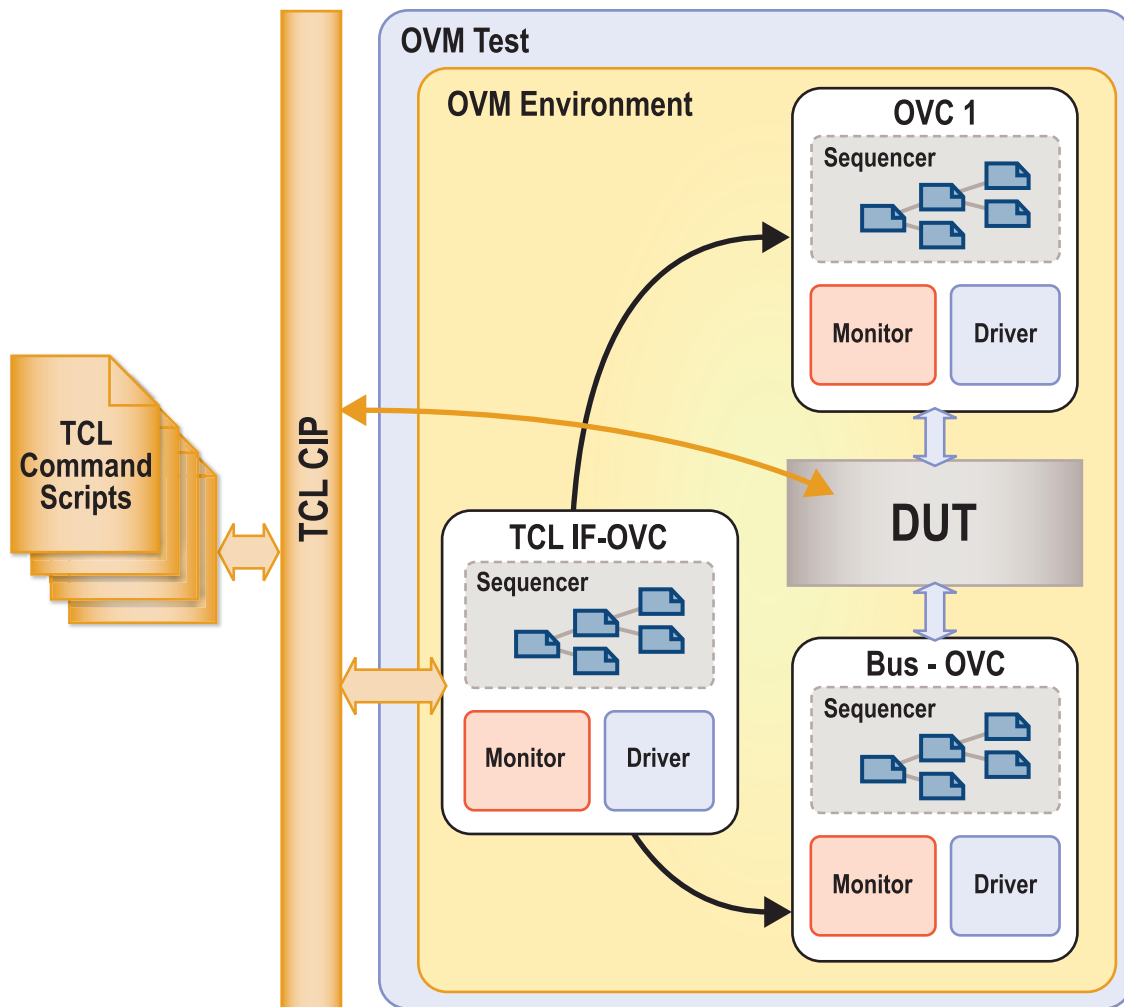


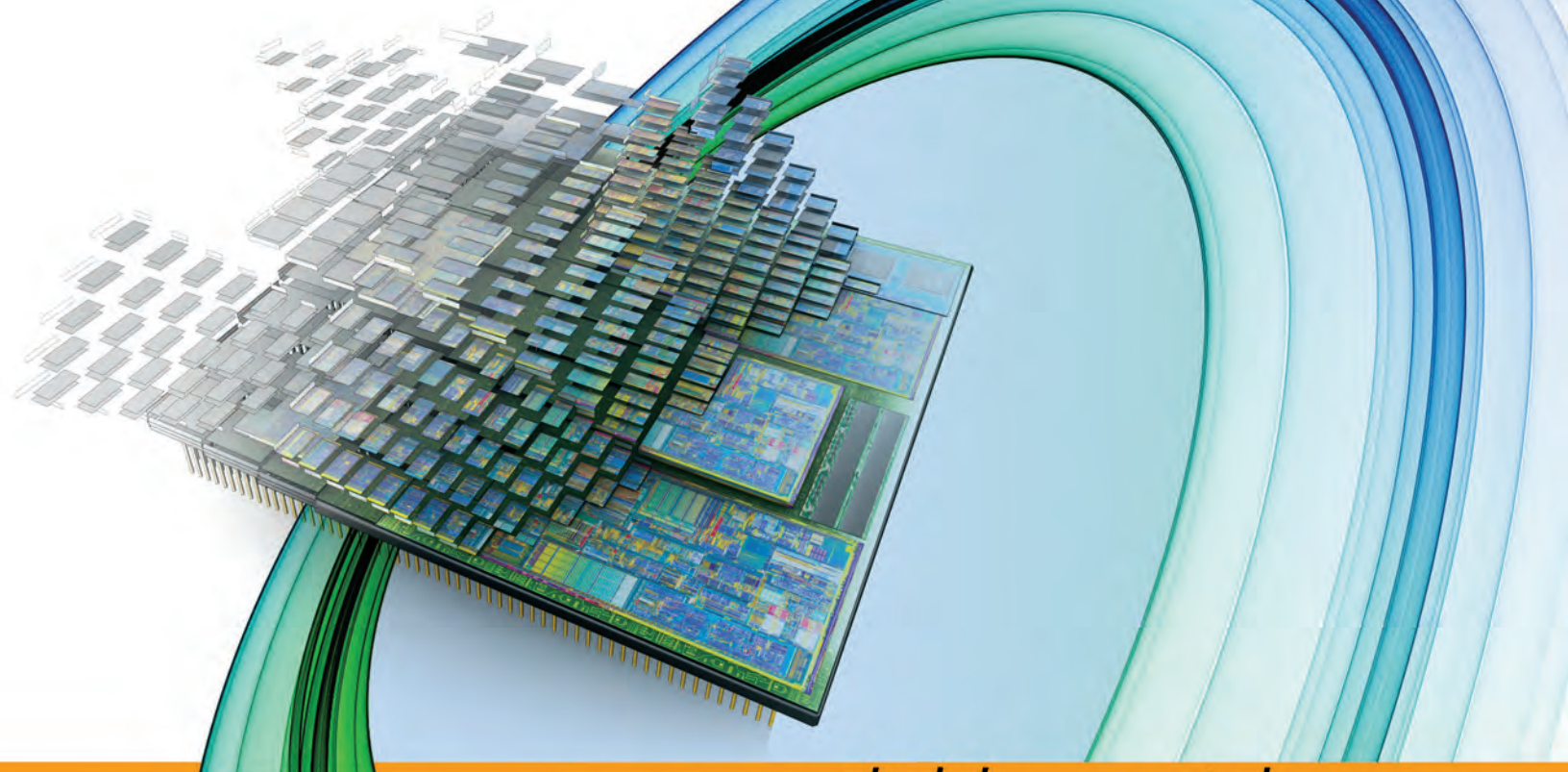
Figure 2 : Overview TCL SV OVM Testbench

## LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
DUT	Design Under Test
HDL	Hardware Description Language
OVCs	Open Verification Components
OVN	Open Verification Methodology
RTL	Register Transfer Level
SV	SystemVerilog
SVA	SystemVerilog Assertion
TCL	Tool Command Language
TCL-CIP	TCL Command Interpreter

## REFERENCES

- [1] OVM SystemVerilog User Guide, Version 2.0.2, June 2009
- [2] Questa® SV/AFV User's Manual, Software Version 6.5c, © 1991-2009 Mentor Graphics Corporation
- [3] Brent Welch, Practical Programming in TCL and TK, Prentice Hall



# *verification* HORIZONS

Editor: Tom Fitzpatrick  
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters  
8005 SW Boeckman Rd.  
Wilsonville, OR 97070-7777  
Phone: 503-685-7000

To subscribe visit:  
[www.mentor.com/horizons](http://www.mentor.com/horizons)

To view our blog visit:  
[VERIFICATIONHORIZONSBLOG.COM](http://VERIFICATIONHORIZONSBLOG.COM)

**Mentor**  
**Graphics**<sup>®</sup>

[www.mentor.com](http://www.mentor.com)