OSVVM and Error Reporting

Jim Lewis, SynthWorks VHDL Training, Tigard, OR, USA (jim@synthworks.com)
+1-503-590-4787

***Abstract*— Testbenches generate error reports and other messages from numerous places. Coordinating these, getting them into a common format, and producing a final report that gives a test passed/failed indication is a challenge. The Open Source VHDL Verification Methodology's (OSVVM) AlertLogPkg package provides a data structure and set of procedures that simplify this situation.**

Keywords—VHDL; Error Reporting for Testbenches ; OSVVM

I. INTRODUCTION

Open Source VHDL Verification Methodology (OSVVM) is a comprehensive, advanced VHDL verification methodology that simplifies implementation of functional coverage, constrained random, Intelligent Coverage Randomization, memory modeling, Alerts, Logs, and transcript files.

OSVVM is implemented as a library of free, open-source packages. OSVVM uses these packages to create features that rival language based implementations in both conciseness, simplicity, and capability.

This paper focuses on the Alerts, Logs, and transcript file capabilities.

II. MOTIVATION

One basic part of a testbench is error signaling and counting. VHDL's assert provides a way to signal errors, but does not count them. As a result, error signaling and counting is commonly done in an ad-hoc fashion. The result is that different models report errors using different mechanisms. At the end of a test, we need an error summary and a passed/failed indication. Generally this means that each testbench has a fair amount of code to collect the different error counts to produce an error summary and a passed/failed indication. This always leaves the potential to accidently leave one count out of the summary. OSVVM provides an Alert and Affirmation capability as a solution to this.

Another fundamental part of testbench is conditionally print information. This information can range from debug messages to passed information regarding aspects of the test plan. VHDL's assert with severity note provides some capability when a simulator's GUI is used to turn it on and off. OSVVM provides a Log capability as a solution to this.

Finally all of the printing done by a test needs to be collected into a single test transcript file. How do we have multiple models write to the same named file? OSVVM provides a test wide file handle in the package TranscriptPkg and all printing in OSVVM uses this transcript file.

III. THE BASICS

The AlertLogPkg separates messages into Alerts (errors) and logs (messages). Alerts generate a message and are counted; logs only print a message. The following is an example of Alerts and Logs.

```
%% Alert ERROR   CpuModel:  No nRdy during CPU Read at 20160 ns
%% Log   PASSED  UART: Expected Value A5 Received value A5   at 31110 ns
```

What differentiates an alert from a log (or VHDL assert) is that when the test is done, a report of the alerts accumulated during a test can be printed. If the test passed, a message of the following format can be printed:

```
%% DONE  PASSED  t1_basic  at 120180 ns
```

Alerts support two modes: simple and hierarchy. In simple mode, alerts are accumulated in a single global resource. With simple mode, if a test fails, a message of the following format can be printed:

```
%% DONE  FAILED  t1_basic  Total Error(s) = 2  Failures: 0  Errors: 1
Warnings: 1  at 0 ns
```

In hierarchy mode, alerts are accumulated separately for each model (or other source of alerts). With hierarchy mode, if a test fails, the message includes error information for each model as shown below:

```
%% DONE  FAILED  t1_basic  Total Error(s) = 21  Failures: 1  Errors: 20
Warnings: 0  at 10117000 ns
%%    Default            Failures: 0  Errors: 4  Warnings: 0
%%    OSVVM              Failures: 0  Errors: 0  Warnings: 0
%%    U_CpuModel         Failures: 0  Errors: 4  Warnings: 0
%%      Data Error       Failures: 0  Errors: 2  Warnings: 0
%%      Protocol Error   Failures: 1  Errors: 2  Warnings: 0
%%    U_UART_TX          Failures: 0  Errors: 6  Warnings: 0
%%    U_UART_RX          Failures: 0  Errors: 6  Warnings: 0
```

IV. ALERTS

Alerts are for reporting errors. Some languages have a similar capability, like VHDL's assert statement, however they typically only print messages and do not provide any sort of summary capability.

An alert is initiated by a call to one of the Alert procedures (Alert, AlertIf, AlertIfNot, AlertIfEqual, AlertIfNotEqual, or AlertIfDiff). Alerts have the levels FAILURE, ERROR, or WARNING. Each level is counted and tracked in an internal data structure. Within the data structure, each of these can be enabled or disabled. A test can be stopped if an alert value has been signaled too many times. Stop values for each counter can be set. At the end of a test, the procedure ReportAlerts prints a report that provides pass/fail and a count of the different alert values.

A. Alerts: Simple Mode, A single global alert counter

By default, there is a single global alert counter. All designs that use alert (or log) need to reference the package AlertLogPkg.

```
library osvvm ;
    use osvvm.AlertLogPkg.all ;
entity tb is
```

An alert is signaled by calling one of the Alert procedures: Alert, AlertIf, AlertIfNot, AlertIfEqual, AlertIfNotEqual, or AlertIfDiff (for files). Alerts can have a level of FAILURE, ERROR, or WARNING.

```
--                    message,          level
When others =>  Alert("Illegal State", FAILURE) ;
. . .
read(Buf, A, ReadValid) ;
--             condition, message,             level
AlertIfNot( ReadValid, "Read of A failed", FAILURE) ;
. . .
--               value1,     value2,       message,              level
AlertIfNotEqual(ActualData, ExpectedData, "Data Actual /= Expected", ERROR) ;
```

The output for an alert is as follows. Alert adds the time at which the error occurred.

```
%% Alert ERROR   Read of A failed  at 20160 ns
```

When a test completes, use ReportAlerts to provide a summary of errors.

```
ReportAlerts( Name => t1_basic) ;
```

When a test passes, the following message is generated:

```
%% DONE  PASSED  t1_basic  at 120180 ns
```

When a test fails, the following message is generated (on a single line):

```
%% DONE  FAILED  t1_basic  Total Error(s) = 2  Failures: 0  Errors: 1
Warnings: 1  at 120180  ns
```

Similar to assert, by default, when an alert FAILURE is signaled, a test failed message (see ReportAlerts) is produced and the simulation is stopped. Alerts generalize this capability with a stop count. A stop count allows a test to stop after a specified number of alerts have been seen. The default for FAILURE is 0 and the default for ERROR and WARNING is integer'right. The following call to SetAlertStopCount, causes a simulation to stop after 20 ERROR level alerts are received.

```
SetAlertStopCount(ERROR, 20) ;
```

Alerts can be enabled by a general enable, SetGlobalAlertEnable (disables all alert handling) or an enable for each alert level, SetAlertEnable. The following call to SetAlertEnable disables WARNING level alerts.

```
SetGlobalAlertEnable(TRUE) ; -- Default
SetAlertEnable(WARNING, FALSE) ;
```

B. Alerts: Hierarchy of Alerts, One alert counter per model

In hierarchy mode, each model and/or source of alerts has its own set of alert counters. Counts from lower levels propagate up to the top level counter. The reason to use hierarchy mode is to get a summary of errors for each model and/or source of alerts in the testbench:

```
%% DONE  FAILED  Testbench  Total Error(s) = 21  Failures: 1  Errors: 20
Warnings: 0  at 10117000 ns
%%    Default            Failures: 0  Errors: 4  Warnings: 0
%%    OSVVM              Failures: 0  Errors: 0  Warnings: 0
%%    U_CpuModel         Failures: 0  Errors: 4  Warnings: 0
%%      Data Error       Failures: 0  Errors: 2  Warnings: 0
%%      Protocol Error   Failures: 1  Errors: 2  Warnings: 0
%%    U_UART_TX          Failures: 0  Errors: 6  Warnings: 0
%%    U_UART_RX          Failures: 0  Errors: 6  Warnings: 0
```

To implement hierarchy mode, AlertLogPkg has a data structure inside of a shared variable. Each level in a hierarchy is referenced with an AlertLogID (of AlertLogIDType). To use alert (or log), a model must allocate an AlertLogID. Then when calling alert (or log), it specifies the AlertLogID as the first parameter in the call.

A new AlertLogID is created by calling the function GetAlertLogID. GetAlertLogID has two parameters: Name and ParentID (of AlertLogIDType). Name is a string value that will be printed as the hierarchy name.

The following example creates three AlertLogIDs: one for the CpuModel (CPU_ALERT_ID), as well as two separate alert counters for counting Data Errors (DATA_ALERT_ID) and Protocol Errors (PROTOCOL_ALERT_ID). CpuModel uses ALERTLOG_BASE_ID (the default if not specified) as the ParentID. DATA_ALERT_ID and PROTOCOL_ALERT_ID use CPU_ALERT_ID as the ParentID.

```
constant CPU_ALERT_ID : AlertLogIDType :=
    --            Name (string value),         Parent AlertLogID
    GetAlertLogID(PathTail(CpuModel'PATH_NAME), ALERTLOG_ BASE_ID) ;
constant DATA_ALERT_ID : AlertLogIDType :=
    GetAlertLogID("Data Error", CPU_ALERT_ID) ;
constant PROTOCOL_ALERT_ID : AlertLogIDType :=
    GetAlertLogID("PROTOCOL Error", CPU_ALERT_ID) ;
```

The AlertLogID is specified first in calls to Alert, SetAlertEnable, and SetAlertStopCount.

```
--             AlertLogID,    Level,    Enable
SetAlertEnable(CPU_ALERT_ID,  WARNING,  FALSE) ;
--               AlertLogID,   Level,   Count
SetAlertStopCount(CPU_ALERT_ID, ERROR,      20) ;
Alert(CPU_ALERT_ID, "CPU Error", ERROR) ;
AlertIf(PROTOCOL_ALERT_ID, inRdy /= '0', "during CPU Read operation", FAILURE);
AlertIfNotEqual(DATA_ALERT_ID,  ReadData,  ExpectedData,  "Actual  /=  Expected
Data");
```

The format of an alert message includes the AlertLogID as shown below.

```
%% Alert FAILURE in CPU_1, Expect data XA5A5 at 2100 ns
```

V. LOGS

Logs provide a mechanism to conditionally print a formatted message. What makes them powerful is that they can be enabled or disabled from different places in the testbench. Log supports levels DEBUG, INFO, FINAL, and ALWAYS. The level ALWAYS enabled, all the others are disabled by default.

In common with alerts, logs support two modes: simple and hierarchy. In simple mode, there is a single set of global controls for enabling DEBUG, INFO, and FINAL. In hierarchy mode, there are separate controls for enabling DEBUG, INFO, and FINAL in each model and/or control point in the hierarchy.

A. Logs: Simple Mode, Global log control

Simple mode is available by default. In simple mode, there is a single set of global controls for enabling DEBUG, INFO, and FINAL. The following provides a short example of setting log controls.

All designs that use log (or alert) need to reference the package AlertLogPkg.

```
library osvvm ;
    use osvvm.AlertLogPkg.all ;
entity tb is
```

In simple mode, levels can be enabled or disabled from anywhere in the testbench by calling SetLogEnable. The log ALWAYS is always enabled, all other logs are disabled by default.

```
SetLogEnable(DEBUG, TRUE) ;
```

The following log will print “A message” when DEBUG is enabled.

```
Log ("A message", DEBUG) ;
```

The format of a log message is as follows.

```
%% Log   DEBUG  A Message at 15110 ns
```

B. Logs: Hierarchy Mode, Each model has separate controls

In hierarchy mode, there are separate controls for enabling DEBUG, INFO, and FINAL in each model and/or control point in the hierarchy.

To implement hierarchy mode, AlertLogPkg has a data structure inside of a shared variable. Each level in a hierarchy is referenced with an AlertLogID (of AlertLogIDType). To use log (or alert), a model must allocate an AlertLogID. Then when calling log (or alert), it specifies the AlertLogID as the first parameter in the call.

A new AlertLogID is created by calling the function GetAlertLogID. GetAlertLogID has two parameters: Name and ParentID (of AlertLogIDType). Name is a string value that will be printed as the hierarchy name. The following creates an AlertLogID for the CpuModel.

```
constant CPU_ALERT_ID : AlertLogIDType :=
    --            Name (string value),         Parent AlertLogID
    GetAlertLogID(PathTail(CpuModel'PATH_NAME), ALERTLOG_ BASE_ID) ;
```

The AlertLogID is specified first in calls to Log and SetLogEnable.

```
SetLogEnable(CPU_ALERT_ID, DEBUG, TRUE) ;
Log (CPU_ALERT_ID, "A message", DEBUG) ;
```

The format of a log message includes the AlertLogID as shown below.

```
%% Log   DEBUG  in CpuModel_1, A Message at 15110 ns
```

Log enables can also be read from a file using the procedure ReadLogEnables.

```
ReadLogEnables("./Test1EnableDebug.txt") ;
```

The file read format is:

```
U_CpuModel DEBUG
U_UART_TX DEBUG INFO
U_UART_RX FINAL INFO DEBUG
```

VI. AFFIRMATIONS

In a high reliability design environment, we need a passed/failed message for each test case. Affirmations do exactly this. When an affirmation fails, it calls Alert and when it passes it calls Log. By default, the Alert Level is ERROR and the Log Level is PASSED. Affirmations support both simple and hierarchy modes.

The following affirmation checks UART expected data (ExpData) with received data (RxData).

```
AffirmIf(UartID, ExpData = RxData, "Expected Value " & to_hstring(ExpData) &
                                   "  Received Data: " & to_hstring(RxData)) ;
```

When the affirmation passes and the PASSED log level is enabled, it prints as shown below. The benefit here is that printing only needs to be turned on for the final reports and need not slow down simulations until all tests have passed.

```
%% Log   PASSED  UART: Expected Value A5   Received value A5 at 31110 ns
```

When the affirmation fails, it prints:

```
%% Alert ERROR   UART: Expected Value A5   Received value 24 at 31110 ns
```

VII. TESTBENCH TRANSCRIPTING

The base layer of alerts and logs is a transcripting utility in TranscriptPkg. TranscriptPkg allows different parts of a testbench to print to a shared (common) transcript file.

TranscriptPkg provides an internal file identifier (TranscriptFile), subprograms for opening (TranscriptOpen) files, closing (TranscriptClose) files, printing (print and writeline), and checking if the file is open (IsTranscriptOpen).

Print prints a string to TranscriptFile when it is open, otherwise, it prints to std.textio.OUTPUT. Writeline does the same thing for an object of type line.

Below is a short program that references the TranscriptPkg, opens a transcript file, and prints using both print and writeline.

```
use osvvm.TranscriptPkg.all ;
architecture Test1 of tb is

  process
  begin
    -- Open TranscriptFile
    TranscriptOpen("./results/Transcript_t1_nominal.txt") ;

    -- Print a string value
    Print("Print 1") ;

    -- Use textio plus WriteLine
    swrite(buf, "Write 1") ;
    writeline(buf) ;
```

```
-- Close TranscriptFile
TranscriptClose ;

-- Print only when the transcript file is open
if IsTranscriptOpen then
  swrite(buf, "Write 1 - transcript open") ;
  writeline(buf) ;
end if;
```

Print works well with VHDL-2008, to_string.

```
Print("Received value: " & to_string(IntVal) & " at " & to_string(NOW)) ;
```

TranscriptPkg also supports a mirror mode in which Print and WriteLine write to both TranscriptFile and std.textio.OUTPUT. Enable mirroring using SetTranscriptMirror.

```
SetTranscriptMirror(TRUE) ;
```

VIII. SUMMARY

Reporting errors, providing uniform and consistent messaging, and printing summary reports does not need to be a complex, tedious task, simply use the utilities provided in AlertLogPkg and TranscriptPkg.

OSVVM has gained notable industry acceptance. It is provided as a pre-compiled library with major VHDL simulator vendors. It has a community of over 1400 members at http://osvvm.org/.

REFERENCES

[1] Jim Lewis, “AlertLogPkg User Guide,” March 2015, AlertLogPkg_User_Guide.pdf

[2] Jim Lewis, “TranscriptPkg User Guide,” January 2015, TranscriptPkg_User_Guide.pdf

[3] Jim Lewis, “VHDL Testbenches and Verification,” September 2015, Training Manual

[4] Jim Lewis, blog articles at http://www.synthworks.com/blog/osvvm/

[5] Jim Lewis, blog articles at http://osvvm.org/