

Bringing Regression Systems into the 21st Century

David Crutchfield
Cypress Semiconductor
New Product Development
Lexington, Kentucky
daac@cypress.com

Thom Ellis
Mentor Graphics Corporation
Design Verification Technology Division
Wilsonville, Oregon
thomas_ellis@mentor.com

Abstract—Today’s verification projects are responsible for verifying some of the largest and most complex designs we have ever seen. Accordingly, the gathering and tracking of development and verification metrics, including coverage and test results, is more important than ever to project success. From figuring out what files are necessary in building a DUT (Design Under Test) and Testbench to knowing what development and verification metrics need to be gathered and tracked, the task can be significant. Like many others, teams at Cypress traditionally had created verification management environments to meet a specific project need. Scripts were either borrowed from other projects or created from scratch and tweaked for the targeted project. Over time this ad-hoc script management often transformed verification environments into an unintelligible mass of interconnected files. Managing such environments requires dedicated resources for each individual project, thus wasting scarce time and money as verification demands continued to grow. This paper will focus on an infrastructure created at Cypress to abstract away file list and metric gathering by providing a uniform front-end shell and back-end database, boosting predictability of testbench creation and metric tracking across multiple projects. Additionally, this paper will discuss various metrics collected and the use of Mentor’s Verification Run Manager (VRM) toolset in gathering metrics, tracking coverage and reducing test suites to quickly and efficiently obtain coverage goals.

Keywords—*functional verification; verification management; verification metrics; verification coverage; test ranking*

I. INTRODUCTION

For years verification management environments within Cypress have been tailored to suit a given project’s needs. This led to largely diverging variations of scripts, each of which had to be managed by dedicated resources. Maintaining these multiple management environments quickly became an overwhelming task given increasing complexity and other demands on verification environments. Additionally, having to interface with multiple management environments created a drag on efficiency, making both resources and projects less portable and eliminating opportunities for shared learning and project reuse. A standard methodology for managing verification was required to provide a uniform interface to users and promote IP reuse. Over time, as enhancements to this standard methodology were made, the entire company benefitted, instead of pockets within the company making improvements. Ideally, each project can benefit from performance enhancements or efficiency improvements, as

well as metrics collection, progress tracking and reporting changes. Furthermore, with consistent metric collection, project regressions can be compared both within and across projects company-wide.

This paper will discuss the verification management infrastructure which includes the following:

- division of labor between Ruby scripts and Questa’s Verification Run Manager (VRM)
- design and testbench tree and file list gathering
- test tree infrastructure and test lists

This paper will also discuss metrics and report generation which includes:

- metrics gathering
- test-level output and reports
- regression-level output and reports
- trending of data

Finally, efficient coverage through ranking, including automated seed generation and test ranking with Questa’s Verification Management tool, will be discussed.

II. VERIFICATION MANAGEMENT SYSTEM INFRASTRUCTURE

A. Division of Labor

Cypress’ VMS, or Verification Management System, is divided between gathering information about the design and testbench through Ruby scripts, and launching tasks through Questa VRM (Verification Run Manager). While Questa VRM was used for launching jobs, it was not required. A suitable tool could be created to schedule, launch and manage jobs in a server grid. Questa VRM was selected to take advantage of its job-management capabilities as well as its automated features such as coverage merging.

The Ruby script handles all front-end tasks and parses through user created files, including design configuration files; testbench files; and tests, compilation and simulation arguments. These user-created files are named *dut.files*, *tb.files*, and *vms.cfg* and will be discussed in detail later. Configuration information can also be passed through a test list and command line arguments as needed.

Once configuration data is gathered it is compiled and placed into intermediate files to be read by Questa VRM. Questa VRM executes based on a RMDB (Run Management DataBase) file. For the purpose of this project the RMDB was designed to be generic; the database configured Questa VRM to read in two separate files. Both files are a set of TCL arrays representing tasks to be executed by VRM. The first file contains compilation tasks while the second file contains simulation tasks. Each task consists of TCL array elements providing detailed information about the task such as design or testbench element name, compilation arguments, test name and simulation arguments, among others. The RMDB is primarily an XML file with TCL constructs. This drove the decision to utilize TCL arrays within the intermediate compilation and simulation task files.

B. Design Tree and File List Gathering

Design networks or trees are described with simple embedded file lists within each design that define what is needed for its inclusion. A design is typically a collection of components or IP that build a subsystem or chip. Within VMS a design is considered to follow this premise, where an IP will be a tree of prerequisite designs making up the whole. With this in mind, an IP will declare what prerequisite designs and extra HDL files are necessary to build its intended function.

VMS standardizes on design description using a file titled *dut.files*. This file consists of commands with arguments that can be various things. The commands within *dut.files* will be interpreted by VMS and used hierarchically to build a design tree which will be flattened to create a design compilation list for verification. An additional VMS requirement is to ensure the DUT is what will be synthesized. Therefore, VMS is used in synthesis list generation, as well. One *dut.files* for an IP will be used to drive all downstream tools, including verification and synthesis, with the appropriate flattened file list. To hierarchically enable VMS, all IPs must have a *dut.files* present. Once they are in place, gathering the necessary files for a given IP is as simple as referencing the design by name. For example, consider the following diagram:

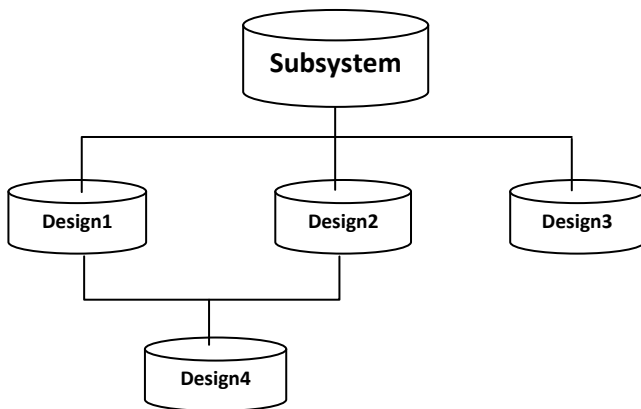


Figure 1: Design Subsystem Tree

In this example Subsystem has three prerequisite designs. They are Design1, Design2, and Design3. Note that

Design1 and Design2 both have the prerequisite design Design4. When *dut.files* is created for Subsystem it will only need to list Design1, Design2, and Design3 as dependencies. If there are HDL files unique to Subsystem, then those would be listed, as well.

Cypress has a defined directory structure that all designs must follow. All designs must be at the same directory level within a workspace and the directory structure for each design must match what has been specified. This provides an expectation that all tools can rely on in finding information. Part of this requirement is that all designs have *dut.files* created and placed in a specific folder for VMS to find. For our example in Figure 1, it is required that each design have an activity folder named *rtl* (i.e. *workspace/design_name/rtl*). In *rtl* will be *dut.files*. Therefore, Subsystem has *dut.files* in *rtl* (*WS/Subsystem/rtl/dut.files*). For the other designs, it would be *WS/Design1/rtl/dut.files*, *WS/Design2/rtl/dut.files* and so on. The contents of *dut.files* for Subsystem are given below:

```

ddc Design1
ddc Design2
ddc Design3
rtl subsystem.v
  
```

Here *dut.files* in Subsystem declares through the command “ddc” that Design1, Design2, and Design3 are prerequisites containing their own *dut.files* to be parsed. Additionally, an HDL file, *subsystem.v*, has been declared with command “rtl”. The significance of “ddc” and “rtl” commands will be discussed more in a moment. Taking the example further, Design1 has its own *dut.files* with the following contents:

```

ddc Design4
rtl something.v
rtl design1_top.v
  
```

Here *dut.files* in Design1 declares that Design4 is a prerequisite and HDL files *something.v* and *design1_top.v* are needed. While this is a simple example, it becomes obvious that complex systems can be created through simple reference to less complex designs.

VMS has been designed to support multiple commands. At this point “ddc” and “rtl” have been introduced. The command “ddc” stands for Design Data Container and tells VMS to look in a specified design’s *rtl* folder for another *dut.files*. The command “rtl” tells VMS to locate an HDL file based on relative location. For instance, if “rtl design1_top.v” is given, then VMS will look in the current folder for *design1_top.v*. However, this command has more significance than just identifying an HDL file. It also tells VMS that the specified file will be used for synthesis. This is necessary because VMS has been configured to build downstream file lists for both verification and synthesis, ensuring that the design verified is what will be synthesized. Another command, “behave,” is used to identify files that are not intended for synthesis but should be included for verification. For a full list of commands supported in *dut.files* see Table 1.

Command	Arg1	Arg2	Purpose
behave	<filename>		The filename provided will be included while building a design verification file list but not a synthesis file list.
ddc	<design>	<arg>	Specifies a prerequisite design. Directs the tool to look in prerequisite design's rtl folder for <i>dut.files</i> . If Arg2 is specified then <i>dut.files.arg</i> will be targeted. If <i>dut.files.arg</i> is not found then <i>rtl/arg/dut.files</i> will be targeted.
gate	<filename>		The filename provided will be included while building a design verification file list but not a synthesis file list.
include	<include path>		Specifies an include path where include files may be located. Translates to '+incdir' option.
rtl	<filename>		The filename provided will be included while building both a verification compilation file list and a synthesis file list.
subdir	<dir name>	<arg>	Specifies a sub-directory based on the current location where a <i>dut.files</i> can be found. If Arg2 is provided then <i>dut.files.arg</i> is processed instead.

Table 1: Command List for *dut.files*

C. Testbench Tree and File List Gathering

Testbench networks or trees are described with simple embedded file lists in the same way that designs are with *dut.files*. For a testbench, the file list is named *tb.files* and is required to exist in the functional verification folder *fnv* under the activity folder *tb* for each design. Much like a design, a complex testbench is a compilation of multiple simple testbench components. Looking again at Figure 1, each design could have testbench information necessary to build the subsystem testbench. If so, then *tb.files* for Subsystem might look like the following:

```
ddc Design1
ddc Design2
ddc Design3
```

While the "ddc" commands are the same as those of Subsystem *dut.files*, they have a different meaning when found in *tb.files* and cause VMS to search in *tb/fnv* of a design for *tb.files*.

For VMS, a testbench is described with a set of commands in *tb.files* that are almost identical to those used for a design. However, there are differences. For instance, "rtl" as used for designs has no meaning with testbenches. Instead, "tb" is given to refer to HDL files. Additionally, as testbenches sometimes utilize C/C++ files, the command "c" is also provided to identify C files intended for C compilation. What type of C compilation (ARM, g++, or gcc) will be further determined based on VMS configuration settings as given

through *vms.cfg*. This will be discussed further in the next section. For a full list of *tb.files* commands see Table 2.

Command	Arg1	Arg2	Purpose
c	<filename>		Specifies C code that needs to be compiled. VMS will use armcc, g++, or gcc depending on <i>vms.cfg</i> settings.
ddc	<design>	<arg>	Specifies a prerequisite testbench. Directs the tool to look in prerequisite testbench's <i>tb/fnv</i> folder for <i>tb.files</i> . If Arg2 is specified then <i>tb.files.arg</i> will be targeted. If <i>tb.files.arg</i> is not found then <i>tb/fnv/arg/tb.files</i> will be targeted.
include	<include path>		Specifies an include path where include files may be located. Translates to '+incdir' option.
subdir	<dir name>	<arg>	Specifies a sub-directory based on the current location where a <i>tb.files</i> can be found. If Arg2 is provided then <i>tb.files.arg</i> is processed instead.
subsystem	<name>		Specifies a subsystem. This command will direct the tool to fetch testbench files for this subsystem from <i>tb.files.name</i> targeted in the current directory.
tb	<filename>		The filename provided will be included while building a testbench verification compilation file list. The directory of filename will automatically be added to the file list with '+incdir'.

Table 2: Command List for *tb.files*

D. Verification Management Configuration

The file *vms.cfg* is used to provide compilation arguments for design and testbench files. It also provides simulation arguments for tests. The default location of this file is in *tb/fnv/tests* for a given design. Simulation mode information can also be found in *vms.cfg* as compilation and simulation arguments are appended with arbitrary mode keys. Additionally, there can be multiple *vms.cfg* files within a testbench's test tree. More will be discussed on test trees later. Table 3 lists most of the *vms.cfg* parameters and their intended uses. Note that all parameters can have <_mode> appended. As stated before, this key is used to identify which arguments should be used for specific modes. A mode can be specified either through the command line or in a user-created test list with the option "-sim_mode". An example might be "-sim_mode RTL," which tells VMS to select *vms.cfg* arguments appended with "_RTL".

Parameter	Description
C_COMPILE_ARGS<_mode>	Arguments to be used when compiling C files found through <i>tb.files</i> .

C_COMPILE_TYPE<_mode>	Specifies what compiler to use when compiling C files found in <i>tb.files</i> . Options are ARM, g++, and gcc. For ARM, ctest.elf will be created for each test. For g++ and gcc, ctest.so will be created and shared by all tests through <i>-sv_lib</i> . VMS will automatically add <i>-sv_lib</i> to simulation arguments for all tests.
C_LINK_ARGS<_mode>	Arguments to be used when linking ARM compiled object files. Compiling and linking is done in one step for g++ and gcc.
COMPILE_ARGS<_mode>	Arguments to be used when compiling files found through <i>dut.files</i> .
FILELIST<_mode>	Declaration of <i>dut.files</i> if the default, <i>rtl/dut.files</i> is not desired. This will be the first <i>dut.files</i> targeted. Multiple <i>dut.files</i> can be listed.
OPT_COMPILE_ARGS<_mode>	Arguments to be used when optimizing a design or testbench before simulation.
SIM_ARGS<_mode>	Arguments to be used when launching a simulation.
TB_FILIEST<_mode>	Declaration of <i>tb.files</i> if the default, <i>tb/fnv/tb.files</i> is not desired. This will be the first <i>tb.files</i> targeted. Multiple <i>tb.files</i> can be listed.
TB_COMPILE_ARGS<_mode>	Arguments to be used when compiling HDL files found through <i>tb.files</i> .

Table 3: *vms.cfg* Parameters

Sometimes it is necessary to further specify or override the *vms.cfg* parameters for given verification runs. This is provided through VMS command line arguments. For most parameters, there is a companion command line option. For instance, “-comp_args” can be used on the command line to give more arguments for DES_COMPILE_ARGS or override them as necessary. Table 4 lists *vms.cfg* parameters and companion command line options. Note that <_mode> is not appended to command line options. This is because mode is specified via command line through “-sim_mode”. Simulation mode can be targeted at run time using this option.

Parameter	Command Line Option
C_COMPILE_ARGS<_mode>	-c_comp_args
C_LINK_ARGS<_mode>	-c_link_args
DES_COMPILE_ARGS<_mode>	-comp_args
SIM_ARGS<_mode>	-sim_args
TB_COMPILE_ARGS<_mode>	-tb_comp_args

Table 4: *vms.cfg* Parameters and Command Line Options

E. Test Tree Infrastructure and Test Lists

VMS is designed to automatically determine which tests to execute without being provided a test list. That is to say, if

configured properly, VMS can figure out what tests to execute. This is accomplished through building a test tree in a format that VMS understands. A test tree is created with inheritance of information throughout branches and leafs to fully describe test arguments unique to each test. Each branch or leaf will have a *vms.cfg* file with information specific to the branch or leaf. Information is inherited down through branches to other branches or leafs. Priority is given to lower levels, meaning that a leaf has priority over a branch with conflicting arguments. If given, command line options will take priority over all levels. Consider Figure 2 below:

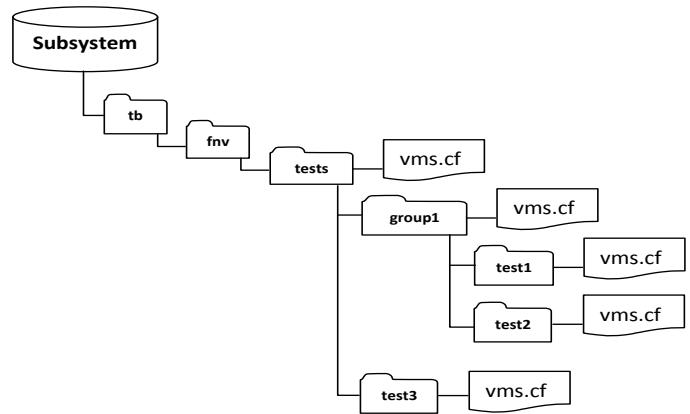


Figure 2: Subsystem with Test Tree

Shown here is Subsystem with a simple test tree under the folder “tests”. There are three tests. Test1 and test2 are grouped under group1. They will both inherit arguments from the *vms.cfg* files above them in “tests” and “group1” for simulation and compilation. Furthermore, test1 and test2 can have unique arguments as defined by *vms.cfg* under the folders test1 and test2. Test3 does not share group1 arguments but does inherit from the “tests” *vms.cfg* arguments. At execution, VMS will search the test tree and identify test1, test2 and test3 as available tests, build arguments properly and launch each.

Before execution, VMS will create a test list of what it thinks should be executed. This test list can be used to target tests within the test tree as desired. For instance, the test list for all tests would look like:

```

group1/test1
group1/test2
test3

```

Arguments can be provided in the test list, as well. For example, one could change definitions without modifying *vms.cfg* files to try out certain configurations. Below is an example test list where only test1 is desired with a new or changed parameter:

```

group1/test1 -sim_args ` -GCLK_FREQ=64`

```

F. Launching VMS

VMS execution is enabled through a Ruby function named `vms_run` at a Unix prompt. VMS is part of the Cypress CAD flow and can be accessed anywhere throughout the company for any given project. If a test tree is configured properly to represent a suite of tests, then a user can simply type `vms_run` and VMS will automatically build a test list from the test tree and start execution. If needed, a user can provide a test list with `-tl test_list` on the command line. In either the test list or via the command line, a user can further configure execution using command line options as shown in Table 4.

Once invoked, VMS will gather information from user input through command line arguments, the design and testbench, and test list command line arguments if they exist and create two intermediate files as discussed before. While processing these intermediate files, Questa's Verification Run Manager will launch jobs accordingly and generate reports as specified through the RMDB file.

Status messages about launched jobs are printed to the Unix terminal. Messages are also stored in a log file to view once a regression has completed (`vms.log`). Additionally, a CSV file, `Status.csv`, is created that will provide tests listed with status and metrics for each listed. An example of a `vms.log` is given in Figure 3 below. Also, an example of `Status.csv` loaded in a spreadsheet editor is given in Figure 4.

```

*** Compiling Design and Testbench Files ***
Compile 1 Pending Thu Jan 9 16:32:02 2014: Design files of DDC system for mode RTL
Compile 1 Started Thu Jan 9 16:32:05 2014: Design files of DDC system for mode RTL
Compile 1 Finished Thu Jan 9 16:32:08 2014: Design files of DDC system for mode RTL
Compile 2 Pending Thu Jan 9 16:32:21 2014: Testbench files of DDC system in subdir /tb/fnv for mode RTL
Compile 2 Started Thu Jan 9 16:32:25 2014: Testbench files of DDC system in subdir /tb/fnv for mode RTL
Compile 2 Finished Thu Jan 9 16:32:31 2014: Testbench files of DDC system in subdir /tb/fnv for mode RTL

*** Preparing Simulation Database. This may take a moment. ***

*** Launching Tests ***
Test 1 Pending Thu Jan 9 16:32:36 2014: 01_rand/01_rand_RTL
Test 2 Pending Thu Jan 9 16:32:40 2014: 02_move/test/test_RTL
Test 1 Started Thu Jan 9 16:32:42 2014: 01_rand/01_rand_RTL
Test 2 Started Thu Jan 9 16:32:43 2014: 02_move/test/test_RTL
Test 1 Finished Thu Jan 9 16:33:06 2014: 01_rand/01_rand_RTL
Status: Passed
Test 3 Pending Thu Jan 9 16:33:06 2014: group2/01_rand/01_rand_RTL
Test 2 Finished Thu Jan 9 16:33:08 2014: 02_move/test/test_RTL
Status: Passed
Test 4 Pending Thu Jan 9 16:33:09 2014: group2/02_move/02_move_RTL
Test 1 Merge Started Thu Jan 9 16:33:09 2014: 01_rand/01_rand_RTL
Test 3 Started Thu Jan 9 16:33:09 2014: group/01_rand/01_rand_RTL
Test 1 Merge Complete Thu Jan 9 16:33:09 2014: 01_rand/01_rand_RTL
Test 4 Started Thu Jan 9 16:33:13 2014: group2/02_move/02_move_RTL
Test 2 Merge Started Thu Jan 9 16:33:14 2014: 02_move/test/test_RTL
Test 2 Merge Complete Thu Jan 9 16:33:14 2014: 02_move/test/test_RTL
Test 3 Finished Thu Jan 9 16:33:39 2014: group2/01_rand/01_rand_RTL
Status: Passed
Test 3 Merge Started Thu Jan 9 16:33:45 2014: group2/01_rand/01_rand_RTL
Test 4 Finished Thu Jan 9 16:33:45 2014: group2/02_move/02_move_RTL
Status: Sim Fatal
Sim LogFile: /disks/moss_disk2/daac/WA/MXGVerificationExamples/top1.2/top/system/tb/fnv/run/tests/group2/02_move/02_move_RTL/02_move_RTL.log
Test 3 Merge Complete Thu Jan 9 16:33:45 2014: group2/01_rand/01_rand_RTL
Test 4 Merge Started Thu Jan 9 16:33:50 2014: group2/02_move/02_move_RTL
*** 30% completion ***
*** 60% completion ***
Test 4 Merge Complete Thu Jan 9 16:33:50 2014: group2/02_move/02_move_RTL
*** 100% completion ***

Summary Test Results Are:
Tests Passing: 3 75.00%
Tests Warning: 0 0.00%
Tests Failing: 0 0.00%
Tests Other : 1 25.00%
-----
Total Tests : 4 100.00% Complete
              : 0 0.00% Not Complete

Aggregate coverage reached on this regression run: 68.6%
    
```

Figure 3: Example `vms.log`

Job ID	State	Status	User	Host	CPU	Mem	Sim Time	Sim Virtual	Sim Working	Launch Time	Completion Time
01_rand/01_rand_RTL	Complete	Passed	NONE	moss	0.0%	0.11	3000	8000	41180	Thu Jan 9 16:32:08 2014	Thu Jan 9 16:32:08 2014
02_move/test/test_RTL	Complete	Passed	NONE	moss	0.11	0.14	24000	81000	42112	Thu Jan 9 16:32:40 2014	Thu Jan 9 16:32:08 2014
group2/01_rand/01_rand_RTL	Complete	Passed	NONE	moss	0.0%	0.07	3000	81000	41360	Thu Jan 9 16:33:06 2014	Thu Jan 9 16:33:06 2014
group2/02_move/02_move_RTL	Complete	Sim Fatal	NONE	moss	0	0	0	87504	24190	Thu Jan 9 16:33:09 2014	Thu Jan 9 16:33:45 2014

Figure 4: Example `Status.csv`

III. METRICS GATHERING, OUTPUT GENERATION AND REPORTING

During simulation, several coverage metrics are gathered for each test. This information is stored in the Unified Coverage Database (UCDB). For verification engineers this is a well known practice that will not be discussed in further detail here. In this section, what will be discussed is the use of test UCDBs to hold additional test metrics. Additionally, the standard placement of test- and regression-level output and reports will be discussed. This section will close with trending reports over multiple regressions.

A. Metrics

VMS captures multiple metrics after a test has finished and stores them in the test UCDB. A list of metrics is given in Table 5 with a short description of each. While these metrics will not appear automatically in Questa-generated reports, they can be accessed at any time from a UCDB through Questa commands or APIs. Further work for VMS is to consolidate regression-gathered metrics into a centralized database making regression data accessible to anyone within Cypress on an as-needed basis. This effort should be complete by midyear 2014.

Metric	Description
Number of Unique Tests	Number of unique tests in test tree. No duplicates due to seeds or argument differences in test list.
Total Number of Tests	Total number of tests for current regression. Includes all seeds and argument variations.
Simulation Mode	The simulation mode or <code>sim_mode</code> "key" specified for each test.
Maximum Runtime	The maximum runtime as specified by the user for each test.
Passing Tests	Total number of passing tests for current regression.
Warning Tests	Total number of tests with a warning for current regression.
Failing Tests	Total number of failing tests for current regression.
Contributing Tests	Total number of contributing tests to coverage for current regression.
Non-contributing Tests	Total number of non-contributing tests to coverage for current regression.

Total CPU Hours	Total number of CPU hours used by current regression.
Test CPU Hours	The CPU hours for each test.
Total Questa Licenses Used	Total number of licenses used by current regression.
Average Resource Wait Time	The average time that tests are waiting on resources. This is the pending state for LSF launch.

Table 5: Metrics Captured by VMS

B. Test Level Output and Reports

In a previous section, the VMS test tree was discussed. VMS uses this tree to gather information about tests and determine what tests to launch. VMS is designed to always be launched from tb/fnv/run. Under the run folder, VMS will replicate the test tree found under “tests”. All tests will be executed from the replicated test tree and all output and report information for each test can be found in its tree branch. For each test, a simulation log file, UCDB file, waveform file (.wlf), debug file (.dbg) and HTML report is generated. Waveforms, debug files and HTML are optionally enabled. The HTML report will have all coverage information for a given test gathered during simulation. Figure 5 below shows a test-level HTML report generated by Questa after finishing a test.

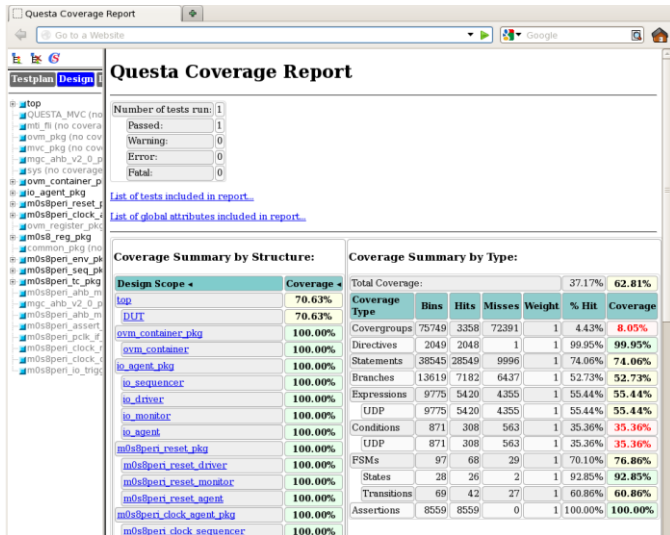


Figure 5: Individual Test HTML Report

For debugging purposes, a file named rerun is created in each test folder. This rerun file can be used to see exactly what command was used to launch a test. Additionally, it can be used to rerun a test without using VMS, if needed.

C. Regression-Level Output and Reports

During a regression run, each test UCDB is automatically merged into a regression UCDB by Questa’s Verification Run Manager and the merged UCDB is placed under the run folder

in the “output” directory. If selected, VMS will also create a regression-level HTML report from this merged UCDB and place it in “report” under the run folder. The HTML report will have all regression-level coverage information as well as regression-level metrics gathered during the regression. Figure 6 below shows a regression-level HTML report generated by Questa after a regression has finished.

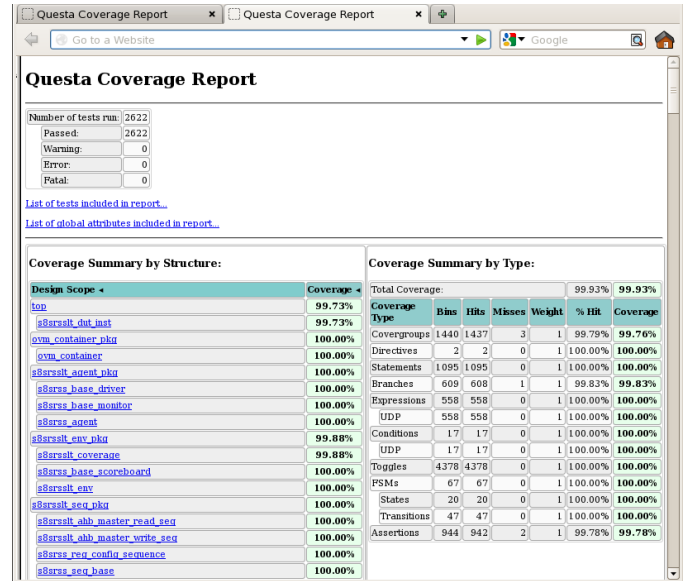


Figure 6: Regression Level HTML Report

D. Trending

Some metrics such as coverage, failing tests, and CPU hours (and many others) are trendable over multiple regressions. It is desirable to see how verification data is converging over time to make predictions on closure and resource planning. When enabled, VMS will generate a trend UCDB and place it in run/output. Additionally, it will create a trend HTML report and place it in run/report. Figure 7 shows an example trend HTML report.

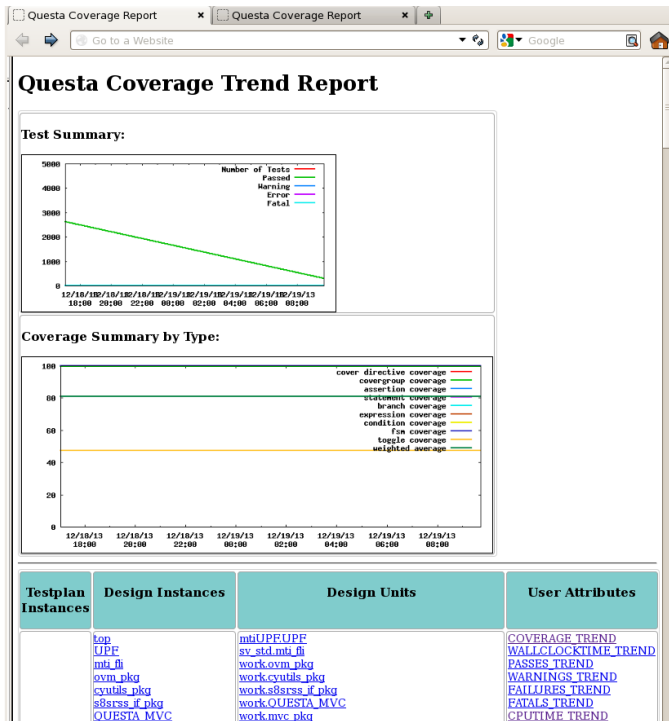


Figure 7: Trend HTML Report

In this example, there were two regressions executed. In the first regression there were 2,622 tests. The last regression was executed after ranking tests and identifying only tests that contribute to coverage. More on this will be discussed later. For now note that coverage did not change between regressions but the total number of tests was reduced from 2,622 to 306.

In Figure 8 the CPU time trend is shown. Here we can see that total CPU time between regressions decreased from 67,855 sec to 8,173 sec.

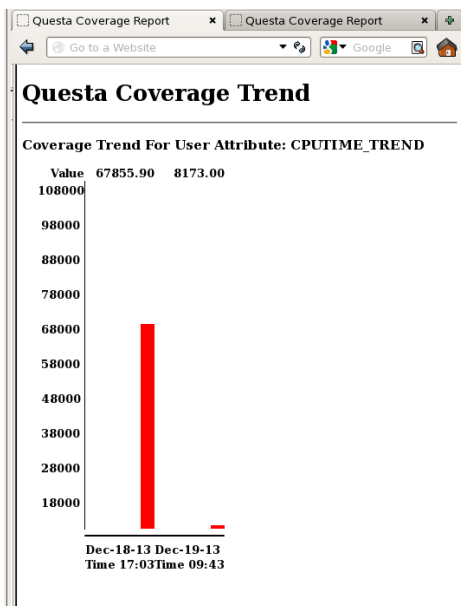


Figure 8: CPU Time Trend

IV. EFFICIENT COVERAGE THROUGH RANKING

A. Automated Seed Generation

VMS provides automation in generating random seeds with multiple options for individual tests. As stated before, VMS utilizes a test tree where specific information about individual tests can be stored. For instance, test arguments such as `sv_seed` can be stored as needed. Within `vms.cfg` VMS provides a parameter named `SEED`. `SEED` can have a list of numbers or number pairs separated by “,” or “;”. If a single number is given, then it will be applied as `sv_seed` in simulation command line arguments. If a number pair is given, then the first number will seed a random number generator and the second number will indicate how many numbers to pull from the random number generator. Each random number will be applied as `sv_seed` in simulation command line arguments. Additionally, “random” can be given as an argument in `SEED`. If so, then the current time will seed the random number generator before pulling a random number. Consider the following example:

```
SEED = [1, 5 6, random 20]
```

In this example, there are 27 total tests to be created with different `sv_seed` values. One test will have `sv_seed = 1`, 6 tests will have random seeds where 5 was used to seed the random number generator before pulling out 6 `sv_seed` values, and 20 tests will have random seeds where the current time was used to seed the random number generator.

In addition to `vms.cfg` `SEED`, command line options or test list command line options can be used to configure seeding. In Table 6 are additional seed control switches that can be applied either through VMS command line arguments or in a user-created test list.

Option	Description
-randseq	Generate sequential <code>sv_seeds</code> starting at a random number.
-rand_seed	Use the argument to seed the random number generator.
-num_seeds	Generate then number of <code>sv_seeds</code> specified.
-sv_seed	Set simulation <code>sv_seed</code> directly with the argument provided.

Table 6: Command Line Options for `sv_seed` Generation

Using “-randseq” allows control over creating sequential random seeds. For instance, the first `sv_seed` will be randomly generated as described before. However, additional seeds will be generated by incrementing the initial randomly generated number. The options “-rand_seed” and “-num_seeds” are the same as number pairs in parameter `SEED` within a `vms.cfg` file. The use of “-sv_seed” is obvious, however, if combined with “-num_seeds”, additional seeds will be generated by incrementing the value given in “-sv_seed”.

Given this automation, random seeds can be targeted through multiple ways. Once seeds are found that contribute to coverage, they can be stored in individual tests through `SEED`

in *vms.cfg* files. This provides an efficient mechanism to randomly test a design and eliminate seeds that do not contribute to coverage that would otherwise waste precious resources.

B. Test Ranking with Questa’s Verification Management

The end goal of verification is to catch all design issues before fabrication. Historically, verification continues long after fabrication as engineers never feel like everything has been covered or they feel that something could possibly have been missed. However, in reality the goal is to cover as much as possible in the shortest amount of time while efficiently utilizing limited resources. Often this effort is aided by leveraging constrained random verification. VMS provides automation in seed generation to address the random goal, though a way to determine what randomly generated seeds are valuable is needed. Mentor’s Questa Verification Management tool suite meets this need.

Test ranking is part of the Questa Verification Management suite and is quite useful in pointing out tests that contribute to coverage. Taking advantage of this tool provides a closed loop between generating random seeds, identifying important seeds and storing them through VMS automation to quickly and efficiently achieve coverage goals. As an example, consider one of Cypress’s internal designs. Before using VMS and Questa’s test ranking feature, we created 2,622 tests to achieve 100% coverage. The seeds were manually picked and replicated for multiple tests. By using Questa’s test ranking feature it was shown that only 306 tests were needed to achieve 100% coverage. Running a regression for 2,622 tests was consuming 18.88 CPU hours to complete. For the 306 tests identified, only 2.27 CPU hours was required, thus allowing Cypress to realize a savings of nearly 90%. Further, the wall clock time reduced from 4 hr 14 min to 49 min. Table 7 is provided below showing results for several designs and the savings realized.

	Original Results			Ranked Results			CPU Perf Savings
	No. Tests	cvg	CPU(T)	No. Tests	cvg	CPU(T)	
Des1	25	99.7%	4 hr	16	99.7%	2 hr	50%
Des2	2622	99.9%	22.4 hr	306	99.9%	2.2 hr	90%
Des3	146	97.4%	16.4 hr	41	97.4%	8.94 hr	46%
Des4	45	99.9%	3.6 hr	28	99.9%	2 hr	46%

Table : Test Ranking Results

Currently, test ranking and seed storage are manual steps that a user must complete. Future additions to VMS will include automating the test ranking process and creating a VMS test list from ranking results.

V. SUMMARY

The Verification Management System developed within Cypress provides a standard methodology for all verification engineers in managing verification. A uniform interface is provided promoting reuse and allowing each project to benefit from performance enhancements or efficiency improvements, as well as metrics collection, progress tracking and reporting changes. Furthermore, with consistent metric collection, project regressions can be compared both within and across projects company-wide. This will position Cypress well for handling the growing complexity and demands of verification environments over time.

ACKNOWLEDGMENT

D. C. thanks Mark Glasser for his input over time while creating VMS. He was a champion within Cypress for building this environment when it initially wasn’t received well. Also D. C. thanks Thom Ellis for helping work through various RMDB coding issues throughout development.