What I Wish My Regression Run Manager's Vendor Knew!

David Crutchfield
Director
Infineon Technologies
8599777555
Lexington, KY 40507
David.Crutchfield@infineon.com

Brian Craw
Lead Principal
Infineon Technologies
8599777572
Lexington, KY 40507
Brian.Craw@infineon.com

Jason Lambirth
Principal
Infineon Technologies
8599777571
Lexington, KY 40507
Jason.Lambirth@infineon.com

***Abstract* - Functional Verification tool vendors typically provide a run manager to launch and track regressions, analyze results, generate reports, and perform a variety of other functions. Solutions range from the very generic, which do not constrain what can be executed, to those that enforce regression jobs to be functional verification centric. High flexibility is ideal but without major configuration, sometimes using vendor's unique take on standard file formats, and hooks to customize report generation and create an intuitive output structure the environment ends up looking like something other than that of a functional test regression. On the other hand, a less flexible solution is easier to setup but does not provide enough tailoring capability when there are actions other than test running needed such as code compilation, customized reporting, job resource analysis, concurrent results checking, etc. Furthermore, run managers provide automation with coverage collection and merging tying a solution to a specific vendor. These issues along with other differences such as emphasis on batch versus GUI usage, license models, UCIS support, and results database access methods lead to increased effort when switching vendors to take advantage of desired features or costs. At Infineon, we have made the painful transition from one vendor to another. This paper will focus on advantages seen in both solutions and what pitfalls users should avoid when integrating. Additionally, it will provide suggestions, based on lessons learned, for which steps or functions should be developed internally versus using the vendor "out-of-box" solution.**

I. INTRODUCTION

Deploying a digital verification run management solution is a must-have to take control of the substantial number of jobs that are required to perform digital functional verification. Vendors provide highly configurable run management tools to tackle this problem. It is left to a company's EDA/methodology group and verification teams to produce and maintain a setup for their users. Lacking a common setup across users and projects leads to a loss in efficiency as each team is left to tackle the job of setting up and maintaining these complex tools themselves. In our history we have had the opportunity to put into place versions of our Verification Management System (VMS) [1] that make use of two predominant run manager tools in the digital functional verification market today: Siemens EDA Verification Run Manager, and Cadence vManager. The issues, solutions, and lessons learned outlined in this paper are a direct result of our experience with setting up, deploying, supporting, and maintaining these tools in our development environment.

II. DEFINITION OF TERMS

DRM	Distributed Resource Manager
LSF	Load Sharing Facility – Workload management platform, job scheduler, for distributed compute clusters
Questa VRM	Questa Verification Run Manager – Siemens EDA regression run manager tool
RMDB	Run Manager DataBase – VRM control file containing a hierarchical tree of Runnables and TCL scripts that allows a user to group tasks together for execution
Runnable	A node in the regression tree. May be a group which includes other Runnables or a task that defines a script to be executed.
User-definable procedure	A procedure written in TCL, in the RMDB file which allows a user to inject custom function at various points of the regression such as at the start or completion of any action
vManager	Cadence verification regression management tool
VMS	Verification Management System – Internally developed tool for gathering design and test bench file information, compilation arguments, simulation arguments and test information, launching each task, and collecting regression information and status
VSIF	Verification Session Input Format – vManager control file containing a hierarchical tree of groups of tasks to be executed

III. REGRESSION CONTROL

The regression management tools' main point of control is through meta-data files. In Questa VRM (Verification Run Manager) this file is called the RMDB (Run Manager Database) file. The syntax is XML with embedded TCL. The Cadence vManager tool uses a VSIF (Verification Session Input Format) file in a JSON/NTF (Nested Text Format) syntax. The VSIF format does not provide support for procs, conditionals, or other programming constructs.

A. Questa VRM RMDB - Overview

For our Questa VRM implementation we provide a single RMDB file. This RMDB file is constructed with Runnables that can be conditionally executed and repeated based on variables. The Runnables may also call user-defined TCL procedures embedded in the RMDB. Our Verification Management System (VMS) script parses regression configuration files that describe the tasks to be executed, i.e., compilation, optimization, simulation, reporting, etc., and converts this information into more easily parsed TCL hashes outputted into intermediate files. Upon execution of VRM the Runnables are executed and these files containing the regression information are read from the user-defined TCL procedures. RMDB variables are set based on this information and Runnables are conditionally executed or repeated.

The flexibility of the RMDB file with its conditional Runnables and embedded TCL scripting enabled us to provide a single RMDB file for all regressions with regression specific information embedded in the intermediate files. This allowed for a common regression environment look and feel for all users through a unique RMDB. While a unique RMDB configured to service all needed tasks can be more complicated, it does help eliminate introduced errors through generation of regression specific RMDBs.

In VRM some procedures are automatically called at key points during the regression. ActionStarted and ActionCompleted are just a couple of these procedures. Upon each task launch the ActionStarted procedure is called. As each task finishes ActionCompleted is called. A feature of VRM is the ability for users to override these procedures and customize them to create their desired verification flow. We made heavy use of these procedures to direct, track, and report the progress of a regression in a way that a user could easily determine which verification steps were being executed. Without customization, the user would only see messages about Runnables and procedures with default names (ActionStarted, etc.). **Figure 1** shows an example of our overloading of the ActionStarted procedure. Upon entry to the procedure the RMDB API is used to determine the action that just started, and a corresponding message is displayed.

The Runnables defined in the RMDB can execute any type of script. They are not limited to only executing simulations. We have defined Runnables to execute compilation, optimization, and simulation scripts, compile C code, run QuestaFormal applications, generate post-regression reports, etc. Furthermore, we have made use of the RMDB structure to define which tasks may be parallelized and which must be serialized. The hierarchical structure of the Runnables defined within the RMDB file also allows for inheritance of parameters down through the tree. This allows for the child Runnables to easily inherit information defined in their parents.

```
proc ActionStarted {userdata} {
  upvar $userdata data
  set action $data(ACTION)

  #### some code removed for clarity ####

  #--Check to see if a simulation was just launched.
  if {[regexp {run_comp_tasks~(.+)preScript} "$action"]} {
    #--Don't do anything for compile pre-scripts
  } elseif {[regexp {run_comp_tasks~(.+)postScript} "$action"]} {
    #--Don't do anything for compile post-scripts
  } elseif {[regexp {run_comp_tasks~(.+)execScript} "$action"]} {
    # report on actual compile tasks - i.e. design comp, tb comp, optimization

    #### some code removed for clarity ####

    if {$filecat=="opt"} {
      if {$ddctype=="des"} {vrunlog [format "Compile %-6s %-8s [RightNow]: Optimizing design of DDC $ddc for mode $compmode" $task_tag "Started"]}
      if {$ddctype=="tb"} {vrunlog [format "Compile %-6s %-8s [RightNow]: Optimizing testbench of DDC $ddc for mode $compmode" $task_tag "Started"]}
    } else {
      if {$ddctype=="des"} {vrunlog [format "Compile %-6s %-8s [RightNow]: Design files of DDC $ddc into $libname for mode $compmode" $task_tag "Started"]}
      if {$ddctype=="tb"} {
        if {$filecat=="gcc" || $filecat=="g++"} {
          vrunlog [format "Compile %-6s %-8s [RightNow]: C files of DDC $ddc in subdir $subdir into $libname for mode $compmode" $task_tag "Started"]
        } else {
          vrunlog [format "Compile %-6s %-8s [RightNow]: Testbench files of DDC $ddc in subdir $subdir into $libname for mode $compmode" $task_tag "Started"]
        }
      }
    }

    set TestStatus($task_tag,state) "Running"
  } elseif {[regexp {simulate/(.+)execScript} "$action"]} {
    # report on simulation tasks

    vrunlog [format "Test %-5s : Sim %-8s %-8s [RightNow]: $TestStatus($testnum,name)" $testentry $seed "Started"]
  } elseif {[regexp {simulate/(.+)mergeScript} "$action"]} {
    # report on merge tasks
    vrunlog [format "Test %-5s : Sim %-8s %-14s [RightNow]: $TestStatus($testnum,name)" $testentry $seed "Merge Started"]
  }
}
```

Figure 1: Overloaded ActionStarted user-defined procedure

B. Questa VRM RMDB – Challenges

TCL code wrapped inside XML syntax makes for a less than rewarding coding experience. XML or TCL syntax errors were difficult to debug due to not always clear runtime errors, and code editor syntax highlighters did not understand how to interpret the TCL embedded in XML syntax. Either TCL syntax or XML syntax can be highlighted but not both.

```
<execScript>
   .
   .
   .
   <command>if {![file exists (%SIMPATH%)]} {mkdir -p (%SIMPATH%)}</command>
   <command>cd (%SIMPATH%)</command>
   <command>  set rerunfile [open (%LOCALRERUNFILE%) w 0744] </command>
   <command>  if {![file exists (%SIMPATH%)/rerun]} {ln -s [exec basename (%LOCALRERUNFILE%)] (%SIMPATH%)/rerun} </command>
   <command>  foreach ctask {(%CTASKS%)} { </command>
   <command>    set cpusuffix  [lindex $ctask 1] </command>
   <command>    if {[file exists ../(%TESTCOMPDIR%)/$cpusuffix] &amp;&amp; ![file exists $cpusuffix] } { </command>
   <command>       ln -s ../(%TESTCOMPDIR%)/$cpusuffix $cpusuffix </command>
   <command>    }       </command>
   <command>  }       </command>
   <command>  set sv_lib "(%SVLIB%)" </command>
   <command>  set command "(%PRESCRIPT%) (%PRESCRIPTARGS%)" </command>
   <command>  echo "Running pre-script $command" </command>
   <command>  puts $prescriptlogfile "Running pre-script $command" </command>
   <command>  puts $rerunfile "$command\n" </command>
   <command>  catch {eval $command} message </command>
   <command>  set eCode $::errorCode </command>
   <command>  set command "qvsim (%NOSTDOUTARG%) -modelsimini (%MODELSIMINI%) (%EXECUTABLE%) -onfinish stop
                          -lib (%LIBS%) -do (%DOFILE%) (%VSIMARGS%) -l (%LOGFILENAME%) $sv_lib" </command>
   .
   .
   .
</execScript>
```

Figure 2: Example of RMDB TCL embedded in XML syntax

Without heavy customization via overriding of the vendor-provided procedures the look and feel of the tool is something other than a functional regression environment. Out of the box the appearance is that of a generic run manager and provides messaging accordingly. While this is inconvenient it does allow for very-specific customization that might not be available if we were restricted to a vendor-provided look-and-feel. A middle-ground suggestion is for vendors to provide reporting “templates” as starting points for customization to lessen the burden of completely re-writing the vendor provided procedures.

We have encountered usage scenarios that lead to degraded performance. The run manager must try and balance between launching new jobs and servicing completed jobs. It is important to keep the compute cluster saturated with jobs to maximize throughput. However, when a large regression consisting of many thousands of short (e.g. <10 seconds) jobs is running the constant interrupts to service the completing jobs prevents the launching of new jobs. The number of jobs running is reduced as they complete but until all the completed jobs are serviced no new jobs, or very few, are being launched. A mechanism to control the prioritization/weighting between launching and servicing jobs would be helpful in these situations.

Over time, to manage all configurations supported by VMS, our custom VRM RMDB file grew to be quite large making support and maintenance increasingly difficult. Maintaining quality and backwards compatibility with a single, large script is difficult and unit-testing is out of the question. In hindsight, a better approach is to move more of the control managed by the embedded TCL code to more atomic, stand-alone scripts, written in a more developer-friendly, modern language (e.g. Python/Ruby), called from the RMDB Runnables. This approach lends itself to a more efficient quality-assurance process while enabling unit-testing. In fact, this is the approach we are currently taking in our VMS implementation that uses vManager. In the future we will re-factor the RMDB to also follow this approach.

C. Cadence vManager VSIF – Overview

The VSIF Nested Text Format (NTF) also allows for a hierarchical approach to regression management. For our VMS with vManager implementation we are generating the VSIF file for each regression. Like our VRM implementation, regression configuration files are read in and intermediate files as well as a VSIF are generated. Independent scripts were then written to build up tool commands and launch them. These scripts are called in the VSIF session, group, and test sections to control the regression. The VSIF allows for user-defined variables to be defined

D. Cadence vManager VSIF – Challenges

The VSIF format is, syntactically speaking, more straight forward than the RMDB format making it easier to generate for each regression. The inherent hierarchical nature of the file makes the structure of the regression more apparent when read. Groups are defined for each “simulation mode” of our environment and below that level test groups are created which can share common collateral, and finally at the leaf nodes test entries describe each test to be executed. Each simulation mode describes a specific “view” of an IP and/or testbench such as RTL, gate, power, debug, etc.

The VSIF syntax assumes that the script executed at a leaf (i.e., test) node is launching a simulation. At first glance this may seem reasonable. However, during our attempts to drive other types of jobs such as compilation scripts (SV and C), report generators, etc., we found this to be a limiting factor. We could place a script to execute these jobs at a leaf node but upon regression completion the test summary results were skewed. Jobs that were not tests were counted as such. Even if we could force the coverage for this “test” to be zero it would still affect the overall test count which leads to confusion when reading reports. We attempted to create groups in the VSIF file that consisted only of pre-group scripts with no “tests” but vManager ignores structures in the VSIF that do not have any leaf nodes, so the scripts were not launched. An enhancement would be to allow VSIF leaf nodes to be somehow tagged as tests which would be included in coverage database merging, results scanning, and metrics gathering. Nodes not tagged as tests would still be executed by vManager but with no built-in assumptions as to their purpose. This would be left to the user.

A similar issue stems from the fact that it is possible in our environment to have multiple compiles per test group. We do not perform a single compilation but launch potentially many compilation scripts to compile various portions of the design and/or testbench independently. This provides control later when only specific portions of the design/tb need re-compiled which is especially helpful in large SoC projects where the long pole of debug turnaround time is re-compilation time. We can lump these compile jobs together in a single pre-group script, but the effect is they are all launched serially not in parallel. We also need to wait for the compiles to finish before launching the elaboration. Therefore, we implemented our own compile job manager script that manages the pre-group compile and elaboration jobs.

Overall, the VSIF format meets most of our needs with the exceptions previously discussed. We are currently generating a single VSIF file per regression defined in our Verification Management System configuration. We may investigate the generation of a VSIF per simulation mode going forward and then rely on vManager/IMC to merge coverage between the sessions. This approach would help by allowing multiple pre-session scripts (for compilation/elaboration) to run in parallel sessions thus avoiding the restriction on not being able to launch multiple compilation jobs in parallel using the native VSIF structure.

IV. BATCH vs GUI

VRM and vManager both support GUI as well as batch modes. For the purposes of this paper, we will focus on the use of these modes for regression running and single regression results reporting. Other use modes such as test-planning and over-time (i.e. multi-session) regression tracking are not considered.

Although both regression managers provide a GUI our Verification Management System (VMS) is primarily built around batch mode. The run manager GUI may still be used but is not required. VMS provides a detailed log report of regression status in real time both to the terminal and a log file. The GUI has its place and is incredibly useful when analyzing results and visualizing large amount of data. These types of analysis mostly take place after a regression has completed. Below are the main reasons we focus on using batch mode over GUI mode for regression launching.

1. Our Verification Management System generates the inputs for the regression manager making it easier to go ahead and launch the run manager and start the regression. There is no need for the user to customize the RMDB or VSIF files. If a user is not using our VMS tool and has provided their own custom control file, then it can make more sense to use the GUI to pick and choose which tasks are to be launched. In an environment where the control file is generated for each regression this customization is done before the run manager is launched.
2. Batch mode gives us more control over the look and feel of the output of the regression which allows us to provide a more vendor and tool agnostic view of a running regression. If we transition between vendors or add more tools to VMS we can ensure the look and feel of the status tracking and reporting is consistent.
3. Not every situation calls for a graphical tool when a batch version more than suffices. For launching and monitoring a regression, minimal status reporting in the terminal is, for the most part, sufficient. A significant amount of our user base would prefer to never open the GUI version of a tool if it can be avoided.
4. GUIs incur a computational overhead and are not always friendly when executed remotely. In some cases, it is not even possible to redirect a GUI from a remote host.
5. We have not made use of a continuous integration setup (e.g. Jenkins) for VMS but it has been on our horizon for some time now. If/when implemented VMS would need to be batch based to be most effective.

V. REPORTING

Both Questa VRM and vManager provide little feedback natively when run in batch mode. This is by design as traditionally large regressions are run nightly in batch mode and little user interaction is required. Therefore, we implemented our own reporting mechanisms.

A. Questa VRM Status Reporting

Since VRM doesn't restrict the user to running functional simulations but rather allows for the execution of any types of scripts, it does not "know" which executables are simulations. Therefore, upon reporting of the results, all executables are counted in the default pass/fail metrics. This can lead to confusion since along with simulations, multiple compilation and optimization jobs, pre-scripts and post-scripts per simulation, and coverage merge jobs may have been executed. A small regression of 10 simulations may result in 40 or 50 "actions" being reported in the regression summary.

With VRM we were able to accomplish customized reporting by adding our messaging to the VRM user-defined procedure (e.g. ActionStarted, ActionCompleted) where desired. Using the VRM API we determine which script was just started/completed and generate a message, if useful, to the user. These user-definable procedures are very powerful and are what enable us to fully customize our VRM implementation. They allow us to only report on actions the user may find interesting such as compilation or simulation job status and filter out noisy notifications about pre or post scripts performing environment setup that we really only want to have status for if there is a failure. Figure 3 shows an example VMS output log showing our custom reporting.

```
vrunlog: Compile m0s0g0t0 Pending  Sun Oct 30 22:59:52 2022: Design files of DDC mydut into work_des_RTL for mode RTL
vrunlog: Compile m0s0g1t0 Pending  Sun Oct 30 22:59:52 2022: Testbench files of DDC mydut in subdir /tb/fnv into work_tb_RTL for mode RTL
vrunlog: Compile m0s0g0t0 Started  Sun Oct 30 23:00:10 2022: Design files of DDC mydut into work_des_RTL for mode RTL
vrunlog: Compile m0s0g0t0 Finished Sun Oct 30 23:00:12 2022: Design files of DDC mydut into work_des_RTL for mode RTL
vrunlog: Compile m0s0g1t0 Started  Sun Oct 30 23:00:20 2022: Testbench files of DDC mydut in subdir /tb/fnv into work_tb_RTL for mode RTL
vrunlog: Compile m0s0g1t0 Finished Sun Oct 30 23:00:26 2022: Testbench files of DDC mydut in subdir /tb/fnv into work_tb_RTL for mode RTL
vrunlog: Compile m0s1g0t0 Pending  Sun Oct 30 23:00:27 2022: Optimizing testbench of DDC optimize_RTL_64 for mode RTL
vrunlog: Compile m0s1g0t0 Started  Sun Oct 30 23:00:30 2022: Optimizing testbench of DDC optimize_RTL_64 for mode RTL
vrunlog: Compile m0s1g0t0 Finished Sun Oct 30 23:00:34 2022: Optimizing testbench of DDC optimize_RTL_64 for mode RTL
vrunlog:
vrunlog: *** Preparing Simulation Database. This may take a moment. ***
vrunlog:
vrunlog: *** Launching Tests ***
vrunlog: Test 1     : Sim 1        Pending  Sun Oct 30 23:00:43 2022: test_tree/intr_set_test/intr_set_test_RTL_135
vrunlog: Test 2     : Sim 1        Pending  Sun Oct 30 23:00:44 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_12345
vrunlog: Test 2     : Sim 2        Pending  Sun Oct 30 23:00:44 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_987654321
vrunlog: Test 1     : Sim 1        Started  Sun Oct 30 23:00:48 2022: test_tree/intr_set_test/intr_set_test_RTL_135
vrunlog: Test 2     : Sim 1        Started  Sun Oct 30 23:00:51 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_12345
vrunlog: Test 1     : Sim 1        Finished Sun Oct 30 23:00:58 2022: test_tree/intr_set_test/intr_set_test_RTL_135
vrunlog:              Status: Passed
vrunlog: Test 2     : Sim 2        Started  Sun Oct 30 23:00:59 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_987654321
vrunlog: Test 1     : Sim 1        Merge Started  Sun Oct 30 23:01:32 2022: test_tree/intr_set_test/intr_set_test_RTL_135
vrunlog: Test 1     : Sim 1        Merge Complete Sun Oct 30 23:01:35 2022: test_tree/intr_set_test/intr_set_test_RTL_135
vrunlog: Test 2     : Sim 1        Finished Sun Oct 30 23:03:05 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_12345
vrunlog:              Status: Passed
vrunlog: Test 2     : Sim 2        Finished Sun Oct 30 23:03:11 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_987654321
vrunlog:              Status: Passed
vrunlog: Test 2     : Sim 1        Merge Started  Sun Oct 30 23:03:44 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_12345
vrunlog: Test 2     : Sim 1        Merge Complete Sun Oct 30 23:03:47 2022: test_tree/sample_irq_dsi_ff00_test/sample_irq_dsi_ff00_test_RTL_12345
vrunlog:

Summary Test Results Are:
Tests Passing: 3     100.00%
Tests Warning: 0     0.00%
Tests Failing: 0     0.00%
Unknown      : 0     0.00%
Tests Other  : 0     0.00%
-------------------------
Total Tests  : 3     100.00% Complete
             : 0     0.00% Not Complete

Coverage on this regression run: 60.98%
```

Figure 3: Example Custom Reporting

B. vManager Status Reporting

vManager does not provide user-definable hooks to the level of granularity that VRM does. It does provide pre/post hooks for each session/group/test level but there is no central location where the information regarding what is currently being executed is provided. Our solution was to create a reporting server which was launched by VMS in parallel to the vManager session. The port and host of this server is provided in the VSIF we generate. All tool commands (e.g. xmvlog, xmelab, xrun, etc.) launched via our generated VSIF are wrapped in our own custom scripts. These custom scripts use the port and host information provided in the VSIF to communicate to the central reporting server to report their status such as "compile/elab/simulation started", "simulation passed/failed", etc. An alternative solution we also investigated was making use of the vManager API (vAPI) to periodically "pull" status information from the session as it ran. However, we found we could not extract test-specific information from the session database without potentially pulling a high-availability license. Even though the license would only be pulled for a short time there would potentially be hundreds to tens-of-thousands of simulations running concurrently in our environment leading to license shortages.

VI. Output Directory Structure

Our experience is that users want to maneuver around their tests' result folders in a terminal. The run managers (both VRM and vManager) force a non-intuitive naming convention on the output directory names.

A feature of VMS is the ability for a user to define a "test tree" which is a hierarchical directory structure that describes the configurations for tests in a regression. Branches in the test tree represent different groupings of tests and nodes in the tree represent one or more tests. It is also possible to have groups within groups. VMS produces test results in a matching directory structure as shown in Figure 3 below. This makes it easier for a user to find tests in the output directory structure. Creating this exact output directory structure natively through VRM or vManager controls is not inherently possible. For the test results folder name VRM does now support a mechanism to name the folder with the actual test name. When VMS was originally implemented this function did not exist. While helpful, we still need to override the full output folder structure to exactly produce the output we desire. In the case of vManager, the test results folders are simply named run_X where X is an incrementing integer. There is no mechanism in vManager to change this behavior.

An example VMS input test tree is shown in Figure 4 followed by examples of the native output directory structure for VRM and vManager in Figures Figure **5** and 6. Finally the desired output folder structure, and the one we have customized the run managers to produce, is shown in Figure 7**.** Figure 5 shows the VRM test results folder for the group1/subgroup1/test1 test. The directory names, i.e., simulate/run_test_tasks~1…, come from the Runnable hierarchy defined in the RMDB file. Similarly, Figure 6 shows the vManager test results folder for the same test. In this case the directory structure at least contains the names of the input hierarchy. However, it is not evident which test correlates to which run_X folder.

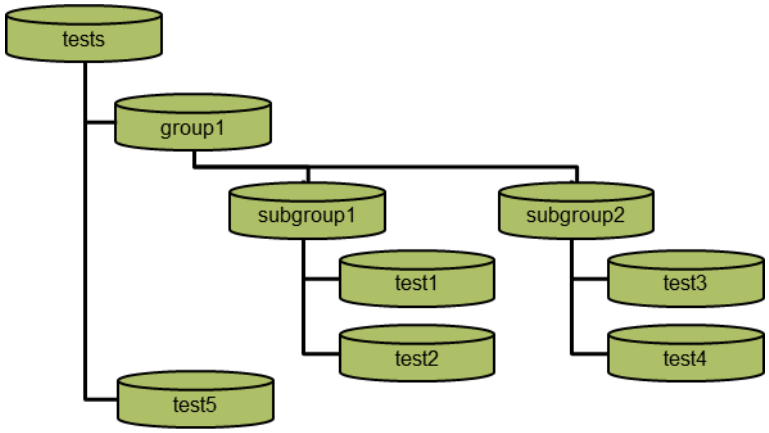

Figure 4: VMS input test tree

```
<VRM_data_path>/simulate/run_test_tasks~1/do_test_specific/get_seed_groups/sim_
seeds~1/simulate qvsim
```

Figure 5: Sample VRM default output directory structure

```
<VMANAGER_regr_path>/chain_0/<mode>/group1_subgroup1/run_1
```

Figure 6: Sample vManager default output directory structure

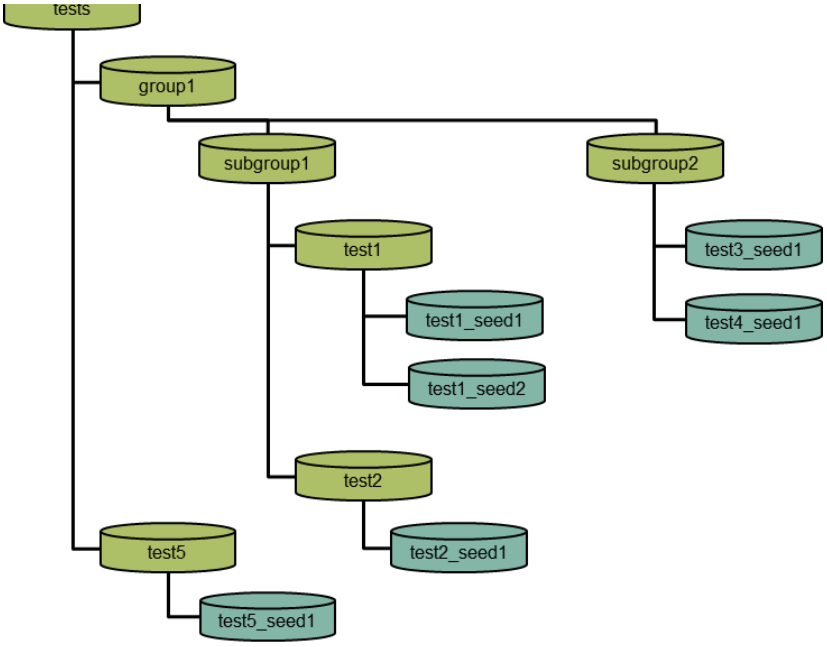

Figure 7: VMS output directory structure

The output directory structure shown in Figure 7 reflects the effect of running multiple seeds of certain tests. For example, the group1/subgroup1/test1 test was run with two seeds. Therefore, the output structure contains two folders under the group1/subgroup1/test1 folder. The test results folder names are appended with the seed. This allows a user to navigate around the output test folder structure in a terminal and know, without opening any log files, which test was run in which folder.

We override the default directory naming by building our own shadow test results tree. The wrappers we place around each command we execute in our VSIF and the Runnables in the RMDB first create and then change directory to the custom folder representing that test before executing.

VII. vMANAGER SERVER

One major difference between the two run managers is that VRM runs as a stand-alone process launching one process per regression, where-as vManager is setup as a central server servicing multiple projects and regressions per project concurrently. There are collaborative benefits to be had by using a central server across users and projects. Regression tracking data can be visualized more easily when multiple users have visibility into each other's regressions. However, there is a cost to this setup in both hardware (servers at each site) and maintenance via an admin needing to continually setup and configure new projects. A better approach in our minds would be to run the regressions as stand-alone processes while still providing a central server that can be used for logging of regressions as needed.

The central server setup also makes it more difficult to extract regression data from the session as it is stored inside a proprietary database requiring, in some cases, a license to access. There is no such restriction when using the VRM implementation. All data stored in the UCDB files is readily accessible with no licensing restrictions.

Finally, when a vManager project is setup on the server a specific version of Xcelium is targeted. If a user wishes to call the Xcelium command directly from their VSIF file, they are restricted to this version of Xcelium. Changing versions to test out a new feature or bug fix requires a server-admin update the project affecting all users of that project which is not efficient. To work-around this restriction we do not call Xcelium directly but instead call a wrapper script which sets the Xcelium version to the version defined in our environment.

VIII. JOB MANAGEMENT

Both VRM and vManager support DRM (Distributed Run Manager) job arrays. This technique can help alleviate the overhead of requesting a grid resource on a per job basis. Job arrays are groups of jobs with similar resource requirements. Multiple jobs are grouped together and launched as a single command. The DRM then divides the group into single jobs as grid servers become available. In the past, due to resource limitations, these features were not used. Grid resources must be shared among multiple business groups with differing resource requirements. It is very difficult to enable job arrays while at the same time limiting potential abuse of the resources by any one person or group. Recently, grid resources and licenses have become less of an issue. We are making use of job arrays in our current implementation using vManager and in the future will re-factor our VRM implementation to do the same.

IX. CONCLUSION

The focus of this paper has been to describe Infineon's experience using two of the industries leading verification run management tools. The paper has outlined challenges we encountered while making use of these tools as engines to drive our internally developed Verification Management System. It is not intended as an exhaustive side-by-side comparison of the two solutions but rather an overview of the biggest challenges we faced and how we overcame them.

VMS was the main digital functional verification entry point for most users at Cypress before that company's acquisition by Infineon Technologies. We are now actively preparing to rollout this tool to a very large user base within Infineon as well. The uniform approach to digital functional verification it provides promotes efficiency and reuse across the company. Therefore, it is important to critically evaluate any tools it makes use of. During our evaluations and implementations of these tools we have had excellent cooperation with both Siemens EDA and Cadence Design Systems. Their help and expertise have been greatly appreciated.

X. REFERENCES

XI.

[1] David Crutchfield, Thom Ellis (2014). *Bringing Regression Systems into the 21st Century*. DVCon 2014, San Jose, CA.