Parallel Debug: A Path to a Better Big Data Diaspora

by Hamilton Carter, Mentor, A Siemens Business

INTRODUCTION
The EDA industry is increasingly avaricious for the benefits of big data. While functional verification has been a producer of big data for several years, paradoxically, big data analysis adoption may not have progressed as quickly as it could have due to a shortage of big data consumers. Most verification engineers have participated in a project where EDA big data was ignored until near the end of the project—if it was gathered at all—at which point there was a mad dash to complete coverage closure.

This article describes a methodology—parallel debug—as well as a supporting Jenkins framework, enabled by the availability of massive processor and disc farms which are commonplace among chip design projects. Parallel debug is an objective, disciplined methodology wherein the engineer changes one and only one aspect of a complex problem based on a hypothesis, and then tests the hypothesis. That is to say, it’s the scientific method repackaged as a debug technique. Many engineers circumvent this technique, making multiple changes to their code at once in (oftentimes vain) hopes of saving time by reducing the number of simulations. Parallel debug provides a methodology for specifying multiple hypotheses; tracking the associated individual code changes via revision control; and—as the name implies—using compute farms to perform all the specified experiments in parallel.

While the debug methodology in and of itself serves to boost the efficiency of verification debug, we hope to capture a second, cultural benefit. As engineers leverage the massive simulation capabilities available, they will be exposed to the beneficial aspects of big data. We hope this exposure will produce a cultural shift leading to swifter adoption, as well as accelerated development of new big data applications.

In the first section of this article, we’ll outline the parallel debug methodology, detailing the specific steps that must be executed by the engineer themselves. In part two, we’ll describe the Jenkins and Questa® Verification Run Manager enabled framework we’re using to automate and deploy this methodology. Part three will describe the known benefits of the methodology, providing a jumping off point for future enhancements. In part four, we’ll briefly discuss some of the opportunities that analysis of the big data we already possess as an industry affords.

PARALLEL DEBUG METHODOLOGY
The methodology is simplicity itself: it’s the scientific method. Applied to debug it goes something like this:

1. Develop one or more testable hypotheses about what’s gone wrong with the design under test (DUT) and/or the verification environment
2. Describe an experiment that can be used to test the hypothesis
3. Execute the experiment
4. Analyze the results of the experiment to determine if the hypothesis was correct
5. Repeat as necessary until the issue is resolved

There are two caveats to the above process. First, hypotheses should not be combined. Each experiment should change only one aspect of the experimental environment. Changing more than one aspect makes it difficult to determine what the root cause of the experimental result was. Second, all the experiments, and their results should be documented and, of course, be reproducible.

A table like the one, upper right, can be used to record each of the hypotheses, details of its associated experiment, and the experimental results.
By maintaining such a table, once the final issue has been resolved, a set of documentation exists detailing the path to the solution. In the event that the experiment was a red herring leading nowhere, the documentation can be used to fall back to an earlier step, and branch in another direction.

This simple methodology, however, can become quite tedious. For example:

- Care must be taken to record notes
- Working one hypothesis at a time is quite time consuming, especially when simulation times run into the minutes or hours
- Making the experiments repeatable requires disciplined revision control habits

Frustration, or simply boredom leads to mistakes, or deliberate departures from the methodology: “Surely changing more than one aspect of the design or verification environment at a time can’t hurt, right? It’ll simply speed things along.” But, it’s exactly these departures from the methodology that ultimately lead to days spent in debug.

Some of these frustrations can be alleviated by parallelizing the debug process utilizing the massive simulation farms available to most chip design projects. Rather than following each hypothesis to its conclusion before starting on the next one, multiple hypotheses can be tested at once.

What might the execution of this process look like? Given a number of hypotheses about the issue at hand, an engineer creates an equal number of revision controlled branches of their verification/design environment—one per hypothesis. The engineer then kicks off a different simulation—each one testing an associated experiment—one of the revision controlled branches. As the simulation results become available, the engineer uses them to evaluate not only the associated hypothesis, but also the other experiments that were executed. Having the results of all the experiments available at once helps to make a much more educated guess if a second round of experiments is necessary. Ideally, the engineer leverages their imagination along with the simulation farm to divide their debug time by the number of hypotheses that can be tested in parallel.

While running the experiments in parallel removes much of the temptation to veer from the methodology, there are still pragmatic irritants. Maintaining clear notes that capture details like what’s been changed and why for each of the hypotheses can be time consuming when we least feel inclined to consume time. Revision control tasks can also become cumbersome and time consuming; in large design projects, initial checkout of a new branch of the development environment can take on the order of tens of minutes. Multiple simulations have to be tracked. Maintaining multiple documentation streams, as well as multiple work flows becomes tedious fairly quickly.

However, if the tedium can be eliminated, the parallel debug methodology provides quick results in the time it would have taken to test only one of the hypotheses using a sequential debug methodology.

In the following section, we describe our system built to automate the process described above.

**BUILDING THE PARALLEL DEBUG SYSTEM**

**Materials**

Our system uses three readily available tools: Jenkins, the Questa® Verification Run Manager (VRM), and the VRM Jenkins plugin. Jenkins, a popular continuous integration tool, provides a framework for defining and launching regression
runs. It’s tightly integrated with revision control systems like SVN and GIT. It also has built-in facilities for storing notes and other artifacts associated with each simulation/regression run.

VRM utilizes an XML format to parameterize, describe, and launch verification jobs that utilize a variety of verification technologies onto a variety of job management systems. The VRM Jenkins plugin displays simulation and regression information captured from jobs launched by VRM as web-based dashboards within Jenkins. Data aggregated from multiple VRM sessions, such as pass/fail and runtime history, is displayed on the associated Jenkins project page. Data specific to a particular VRM session is displayed on the underlying Jenkins Build page. See Ellis, T. for a more general discussion of using Jenkins and VRM together with the Jenkins/VRM plugin.

Construction
The Jenkins web-based UI is organized hierarchically into folders, projects, and builds. Folders within Jenkins contain one or more projects. Each project corresponds to a single revision-controlled branch of a code repository.

The Jenkins folder/project hierarchy

Projects in turn contain Builds. A Jenkins build defines a script-based process that can be launched on the Jenkins server. The script can be written in your favorite scripting language; our build scripts—consisting of calls to VRM to start a group of one or more simulations—have been written in Bash.

Pro Tip: Everything Changes, even build scripts: Rather than writing our build scripts directly into the Jenkins UI, (a valid and widely accepted model), we stored them in the same revision controlled repository as the design and verification environment. Then, we simply call the script from the Jenkins UI. Since Jenkins first updates itself to the latest revision of a branch, and only then executes the build script, when we want to recreate a build we just point the Jenkins project at a branch copied from that revision. We’re guaranteed not only to get the exact same design and verification environment but also that Jenkins is performing the exact same build process steps that it originally did.

After a project has been built, Jenkins captures and stores pertinent metrics. Jenkins automatically stores the pass/fail status of each build, as well as the SVN change history between builds. The VRM plugin also archives a number of verification-centric metrics within the Jenkins database. These metrics include things like the pass/fail status of each simulation, the amount of compute time each sim consumes, and a pointer to the log file for each sim.

How Jenkins fits into the methodology
The Jenkins data model described above is perfectly suited to the debug methodology described in section one. We represent the issue to be debugged as a Jenkins folder. Each Jenkins project organized under the folder represents a single hypothesis. In the event that a hypothesis leads to more experiments that build sequentially on the original, the builds within a Project correspond to each of these additional experiments. In the event that the findings of an experiment result in multiple additional experiments, then a new nested folder is created. The projects within that folder correspond to the newly minted experimental group.
An example of the described organization is shown in the diagram below:

Automating the parallel debug methodology
Our system uses the tools described above to automate the parallel debug process. The user interacts with the system using a setup script `create_ensemble.sh` and the Jenkins web-based UI. `create_ensemble.sh` performs the following steps to build the infrastructure shown in the figure below:

- Create the individual experimental branches
- Create the Jenkins experimental ensemble
- Parallelize the initial revision control system repository checkouts

These steps are described in detail in the following paragraphs.

Creating the individual experimental branches
When called with the syntax `create_ensemble.sh COUNT BRANCH SUBBRANCH` the setup script first creates one revision control sub-branch for each of the COUNT experiments specified by the engineer. Each branch is copied from BRANCH and is labeled `branch_subbranch_i`. A repository corresponding to each branch is checked out into a corresponding work area. These checkouts are submitted in parallel to a job control system.

Creating the Jenkins Experimental Ensemble
Next, the Jenkins REST API is called to launch a project that sets up the Jenkins side of the system. The project’s build script has write permission for the underlying Jenkins directories. It creates a folder named `branch_subbranch` corresponding to the debug issue and then copies COUNT number of generic simulation projects under the folder. Each project is labeled `subbranch_i` and points to its associated revision controlled branch (created in the previous step). It then calls the Jenkins REST API to reload Jenkins from disc, ensuring that the newly created folder and projects will be visible on the Jenkins web site.

Tucking initial checkout times under the covers
Finally, the Jenkins setup script uses the REST API one last time to kick off a no-op simulation build in each project. This build triggers Jenkins to perform its initial checkout of the design/verification environment. Because all of the simulation builds are kicked off in parallel via Jenkins’ REST API, we only incur the time of one checkout (under the assumption, of course, that there are enough machines to farm out all of the checkouts in parallel).
THE DOUBLE-DOWN WORK AREA SYSTEM
You probably noticed that we have two work areas for each experiment: one for Jenkins, and one for the engineer. This was originally a pragmatic design decision. There were simply too many issues to fight in keeping a single work area; either Jenkins had to have write permission into the engineer’s disc space and SVN account, or vice-versa. Having separate work areas for Jenkins and the engineer was a simple workaround.

However, what started out as a workaround has become an important part of the methodological framework. Jenkins will only execute a simulation containing the engineer’s most recent changes if the engineer has committed them to revision control. In this manner, the revision control of each change made during a series of experiments is enforced.

Back to the Engineer
Everything else is handled by the engineer, Jenkins, and VRM. Our VRM scripts accept a list of test names as input. The only thing the engineer has to initially customize (besides the changes required for the experiment itself) is the name of the testcase they would like to run. Once their test list and experimental changes have been committed to revision control, they simply start a build of the associated experiment. Jenkins kicks off the build script, which calls VRM. The underlying verification engines (simulators, formal tools, emulators, etc.) are launched by VRM, and their results are in turn captured, organized, and displayed by VRM and the plugin.

The next section describes a typical debug session using the methodology, facilitated by the system, outlined above.

A DAY IN THE LIFE
The Debug Flow Chronicled
Arriving in the morning, you find that a newly discovered design issue has been assigned to you. You review the description of the issue, crack open a debug spreadsheet, and copy the issue description into it. Ruminating over your first cup of coffee and donut (more engineers should eat donuts), you jot down several hypotheses, guesses, impressions (call them what you like, but record each one on a separate row) relating to what may have been the root cause of the issue.

At this point, you start create _ensemble.sh with COUNT equal to the number of rows in the spreadsheet. The compute farm comes into play, creating all the necessary revision controlled work areas for both Jenkins and you, as described in the system build section.

While that’s running, and as you launch into your second cup of coffee, you describe what should be changed in the design or verification environment to produce an experiment that can either prove, disprove, or perhaps—in complex cases—just shed more light on your initial guess.

With the experimental procedures outlined, the last non-automated documentation preservation step begins. You copy the description of the debug issue to the associated Jenkins folder’s description field. You perform the same operation for each of the spreadsheet rows, copying your text into the Jenkins project description field for each experiment.

The Jenkins folder’s description field
The Jenkins folder corresponding to the debug issue, and its contained collection of projects, now hold the same data as the original spreadsheet.

Now the real engineering work begins in earnest. The necessary files are modified and committed for each experiment in each associated branch’s work area. Jenkins launches each of the experiments. The results for each experiment are displayed in the Jenkins UI.

As the initial results return, they either confirm or disprove your conjectures. If further debug is required, you use the data returned by the initial experiments to plan the next round. Artifacts documenting the progression of these experiments are stored in the results and notes contained in each project and its builds.

In this manner, you capture your entire debug thought process as an annotated and—just as importantly—executable workflow.

**BENEFITS OF THE METHODOLOGY AND THE ENABLING SYSTEM**

**The obvious: rigorous, controlled change management**

Perhaps the most obvious benefit is the ability to deliver on and enhance the promise of revision control. This stems both from the rigor that the automated framework imposes as well as Jenkins’ easy-to-use view into the revision control system.

To try new experiments, thanks to the double-down work area system described in section two, an engineer has to commit their changes before they can be picked up by Jenkins and simulated in a build. This simple rule forces the engineer to track changes made for each experiment; changes that any of us—as humans—might have decided to shrug off because well, “it’s an obvious change,” or because “I’ll remember.” With these excuses disallowed, every step of an experiment is available for historical review.

Engineers, and the project team as a whole, are also far more likely to benefit from the revision control system as it’s encapsulated in Jenkins. There’s no need to remember the various commands for retrieving the change log for a file. Likewise, there’s no system of note-keeping to define to make sure that revisions are cataloged. Everything is stored in an easily accessible format embedded with the specific builds contained in the Jenkins project.

**Persistent experiment notes**

With artifacts that annotate the debug process, engineers do not have to reinvent the wheel when they find themselves wondering why they made a change to the code. It’s not the information in and of itself, but rather the information combined with the convenience and the rigor of the enabling system that make the real difference. Sure, you could have run SVN log to find out what changed during the time you were working on a give portion of the project, but the process of remembering dates, branch names, and SVN commands is just daunting enough to convince many engineers not to bother.

**Executable thought processes**

The system allows you to recreate debug steps exactly as they were initially found in the wild. To begin working on an issue from any point in a debug tree, the user just needs to make a new copy of the repository using the revision number at that point. Then they can read the revision number directly from Jenkins.

1. `svn copy -rxxx initial_branch initial_branch_rxxx`
2. `create_ensemble.sh 1 initial_branch_rxxx sandbox`
The above two steps first recreate the simulation environment as it appeared in the debug revision denoted by Jenkins, and then start a build in the new folder `initial_branch_rxxx_sandbox` so that it can be further explored.

**DELIVERING ON THE PROMISE OF BIG DATA**

So, now you’ve seen it, but how does this automated methodology get us from enhanced debug efficiency to a cultural shift using big data? Deploying this process within a project makes engineers more aware of what they can accomplish with big data in two ways. First, it shows them that their massive compute farms aren’t just for regressions. Second, the VRM plugin creates a constant awareness of the magnitude and variety of data that is easily accessible from their existing simulation runs. These two newly opened vistas have inspired a number of follow-on methodological tool flow applications. A few of the promising horizons we see are:

**Drive to coverage closure like you have a roadmap** by capturing code coverage—and for that matter functional coverage—not at the end of a verification project, but at the start, and continuously throughout the project. Because all verification jobs are kicked off using Jenkins, coverage information is always readily available through the VRM plugin. Engineers can drive to closure from the start of the project rather than pray for closure at the end.

**Treating the patient, (er project), while it’s still alive; or how to avoid postmortems:** Now instead of waiting till the end of a project to try to figure out what went right and wrong, engineering teams can use the available data to constantly adapt and improve. For example, the captured SVN data from Jenkins can be used to judge the impacts that design and verification changes are having on the project. Additionally, both historical and contemporary coverage information can be used to determine where effort should be applied next and where things are beginning to slip.

**Tuning for coverage:** Optimizing test cases for coverage is an activity that seems obvious. I mean all the coverage driven verification methodologies proclaim its importance, but how many projects actually worry about it right from the start? Once again, the double edged sword of being able to easily kick off multiple jobs and being aware of the data available is incredibly powerful. As engineers execute simulations, they’re provided with information about how effective a set of constraints is at generating coverage, how it compared to other sets of constraints, and how much that coverage costs with respect to simulation run time.

There are many other applications of big data just waiting to create better informed, more efficient verification teams. We’re looking forward to seeing what the future brings!

**REFERENCES**

VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

30 Video Courses Available Covering
- UVM Debug
- Sequential Logic Equivalence Checking
- Portable Stimulus Basics
- SystemVerilog OOP
- Formal Verification
- Intelligent Testbench Automation
- Metrics in SoC Verification
- Verification Planning
- Introductory, Basic, and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- PowerAware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 8100 topics
Verification Patterns Library

www.verificationacademy.com