SYNOPSYS®

WHITE PAPER

Using Machine Learning to Automate Debug of Simulation Regression Results

Author

Rob van Blommestein
Product Marketing

Background

In the history of semiconductor verification, few advancements have had more impact than the introduction of regressions for simulation test suites. At one time, the concept of verification barely existed. Designers were expected to do some ad hoc testing of their designs, typically by manually applying some inputs, running a short simulation, and looking at the resulting waveforms. As designs grew larger and more complex, it became important to formalize this process. Verification engineering emerged as a distinct discipline from design. The verification engineers hand-wrote scripted tests for specific features of the design, ran them in simulation, and debugged the failures. They listed the features in a verification plan and checked them off as they were tested. When all features were checked, the chip was taped out.

Regressions entered the picture because running each test once was not enough. The verification team found that tests that had previously passed frequently failed when rerun later in the project. There were several causes for this effect. To accelerate the schedule and shorten time to market (TTM), the design and verification efforts overlapped. Portions of the design added later often changed the behavior of portions previously tested. Tests also failed when fixes for one design bug introduced new bugs or caused ripple effects. Refining the design as it was being implemented (synthesized, placed, and routed) or outfitted with design for test (DFT) features also tended to create new bugs. Finally, the chip specification itself changed many times throughout the course of the project, causing some tests to be outdated or leading the design and verification teams to get out of sync.

For all these reasons, verification engineers collected all passing tests into a test suite and ran them in regression runs on a regular basis, often nightly when the available simulation and compute resources made that possible. This remains the case today. Constrained random stimulus generation has automated and reduced the manual effort in test creation, but the tests are still collected into suites and simulated in regressions. For much of a project, at least a few previously passing tests fail in every regression run. Sometimes every test fails, especially when major changes are made to the verification environment or testbench. Debugging these failures is largely a manual effort, with costly impact on resources and TTM. This white paper presents a novel solution: the use of artificial intelligence (AI) and machine learning (ML) techniques to automate the debug of regression results.

Challenges for Regression Debug

As they write new tests or modify constraints to generate new random tests, verification engineers expect a certain amount of manual debug effort. Any design has lots of bugs to be found and fixed, so it is normal for the first run of many new tests to fail because they have uncovered design bugs. Of course, verification engineers make mistakes just as designers do, and so many tests fail because of errors in the tests or testbench. Figuring out the causes of the failures and fixing them requires manual work, but it is precisely what verification is all about. The mindset of finding this a fun challenge and cleverly diagnosing bugs is part of what defines a verification engineer.

Failures in regression runs are a different matter entirely. New tests are generally debugged in local "sandboxes" and added to the project regression suite only after they pass. Ideally, once tests have been included in regressions the verification team is done with them. This never happens, but the team hopes that regression failures are infrequent since they interrupt progress on creating new tests and are much less fun to debug. In some ways, every regression failure is a disappointment since it represents a step back in the project timeline and requires revisiting parts of the verification plan previously considered done. It's discouraging for verification engineers to check the results of an overnight regression run and see failures.

Unfortunately, for all the reasons mentioned earlier, regression failures are common. For much of a project, they are more the rule than the exception. As shown Figure 1, regression failure debug is usually a manual process. After the latest changes to the design and verification code are checked in, the regression is run, and some tests fail. Hundreds or thousands of failing tests are not uncommon, presenting a daunting task to the verification engineers. They start by manually examining the simulation log files and analyzing what went wrong. It is often possible to categorize the failures and sort them into "bins" based on the type of error reported. The bins then need to be triaged to determine whether the problem is more likely to be in the design or the testbench, and which blocks or components are the primary suspects.

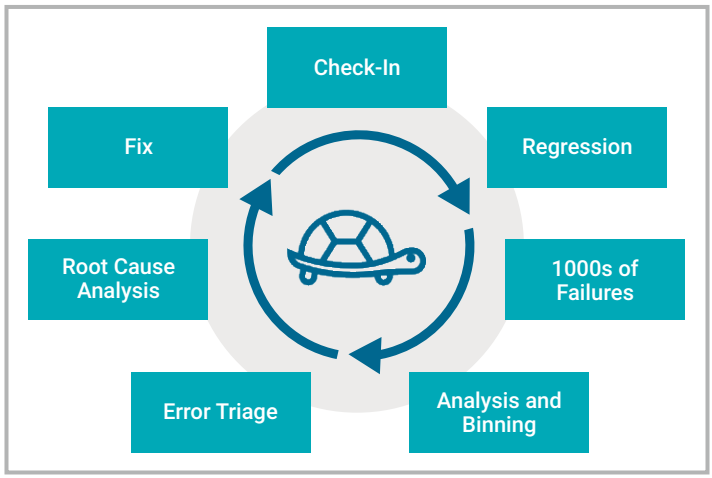

Figure 1: Regression testing loop with manual steps

At this point, debugging passes to the verification engineers or designers who own the suspect code. They perform root cause analysis (RCA) to try to pinpoint the actual bug triggering the test failure. This process is manual and iterative for all the logs from all the failing regression tests. It consumes valuable project time, ties up expensive resources, and is error prone due to the manual nature of the analysis and triage. Incorrect binning means that multiple failures due to the same bug are sometimes triaged multiple times rather than just once. Manual triage means that a failure is frequently passed on to several design or verification engineers before the root cause is finally found and fixed. Chip development teams have been clamoring for a better way to manage and debug regressions.

Machine Learning to the Rescue

Only recently has technology been available to automatically analyze, bin, triage, probe, and discover the root causes of regression failures. ML is the key enabler for these capabilities, and it's not hard to see why. Manual regression debug relies heavily on the experience of the verification engineers. Over time, they develop a sense of what types of failures occur and how they can be binned appropriately. Years of triage help them more accurately determine the most likely sources for the failures and to assign them to the right design and verification engineers for root cause analysis and fixes. Given the enormous amount of information gleaned from thousands of regression runs on a project, ML can automatically gain an AI version of this experience and apply it for faster and more accurate debug. Figure 2 shows how three stages of the regression loop are accelerated and automated with ML techniques.

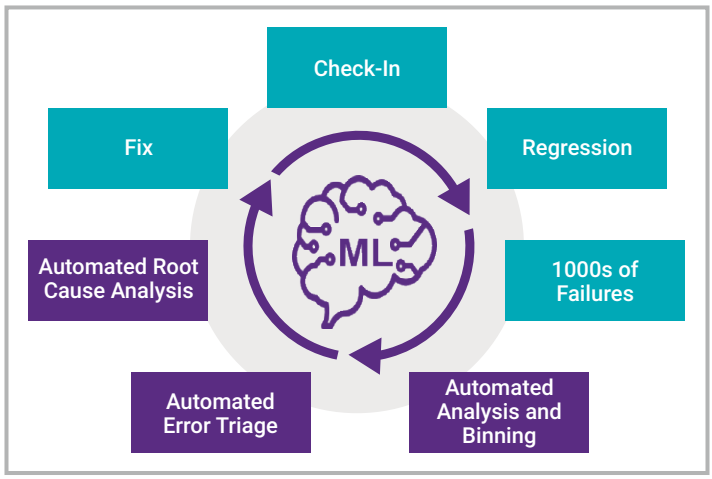

Figure 2: Regression testing loop with ML assistance

A solution enabling this much more efficient loop is available today. The Regression Debug Automation (RDA) capabilities in Synopsys Verdi® Automated Debug System automatically discover the root causes of regression failures. RDA classifies and analyzes raw regression failures using ML and identifies root causes of failures in the design and testbench. RDA focuses on automating the regression log analysis, binning, triage, and root cause analysis to reduce the tedious and manual effort associated with the typical regression flow. RDA automation helps the users find, understand, and fix the bugs much faster than previously done before, improving the overall debug effort by 2X or more.The overall RDA flow is shown in Figure 3.

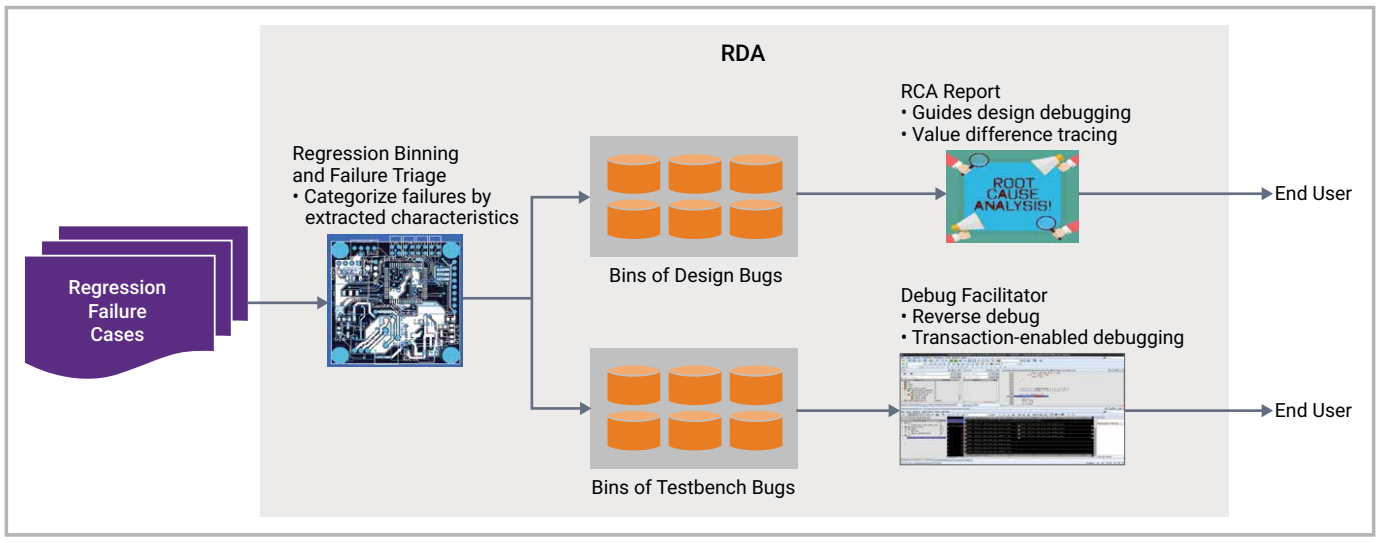

Figure 3: Overall Verdi RDA flow

The traditional manual steps of binning and triage on the regression failures are automated, running after the regression, usually overnight. Thus, the debug environment is set up at night without any need for user action or intervention. Automated RCA is performed on any bugs traced to the design, and reports provided to the designers make it much easier to determine the exact cause of the failure. Reporting which signals had different values between passing and failing test runs focuses the debug effort on the right part of the design. For bugs in the testbench code, transaction-aware debug and the ability to move both forward and backward in the simulation timeline makes it much easier for the verification team to understand and resolve the source of test failures.

Components of the Solution

Verdi RDA incorporates numerous powerful techniques and technologies to automate and accelerate regression debug. The process starts by collecting data from the regression run, including simulation log files, value change dump (FSDB) files, and compiled simulation databases with the design and Universal Verification Methodology (UVM) testbenches. The collected data feeds into the regression binning application, which analyzes the log files for failures and sorts those failures into like categories. This step uses unsupervised ML to mine relationships among the verification log failures and bin the results. It takes advantage of UVM-based messaging, user-defined rules, verification intellectual property (VIP) log binning, and CPU-based design (instruction set) binning. It is simple to set up and thus easy to use. This process has been shown to be 90% accurate and reduces the overall triage time. As shown in Figure 4, the results are brought into Verdi's RCA Manager for review by the design and verification engineers.

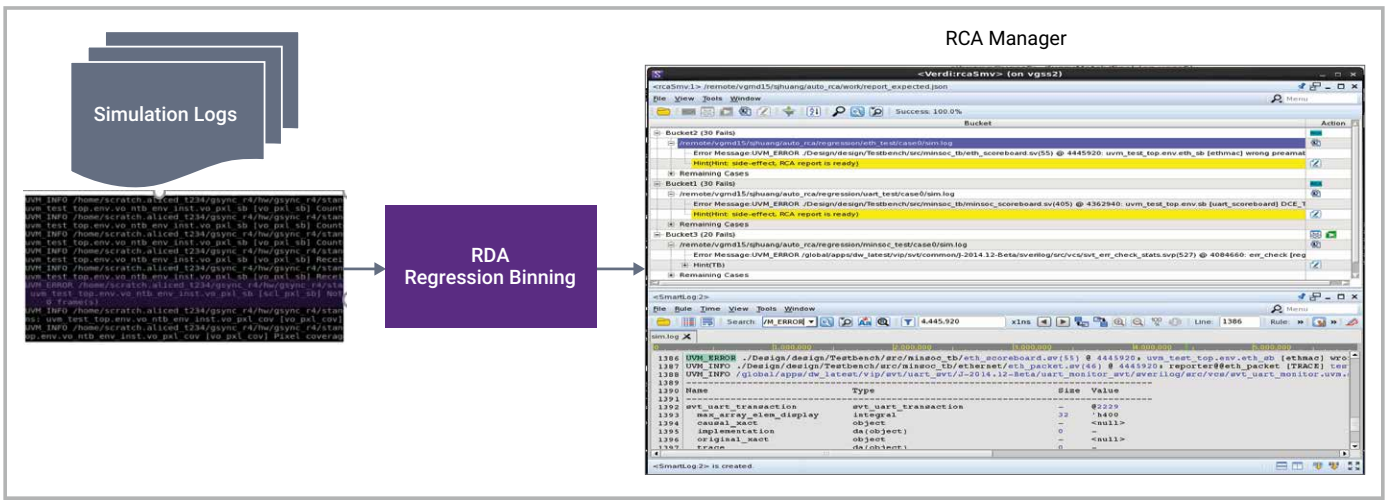

Figure 4: ML-based regression binning in Verdi RDA

After binning, RDA performs failure analysis and triage. It takes the bins of failures and determines whether the issues are from the design under test (DUT) or the simulation testbench based on the characteristics of the failures. One of the reasons that Verdi RDA is so effective is its application of multiple technologies to find the root cause of failures in both the design and the testbench. The DUTRCA approach identifies failures from the log files and then compares the values of signals from passing and failing tests to isolate failure points that differ near the test errors. DUTRCA uses time-based roll back mechanisms and its TraceDiff capability to automatically narrow down the cause of the error in the DUT. As shown in Figure 5, Verdi provides visualization to show the RCA path along with the signal value changes in the design.

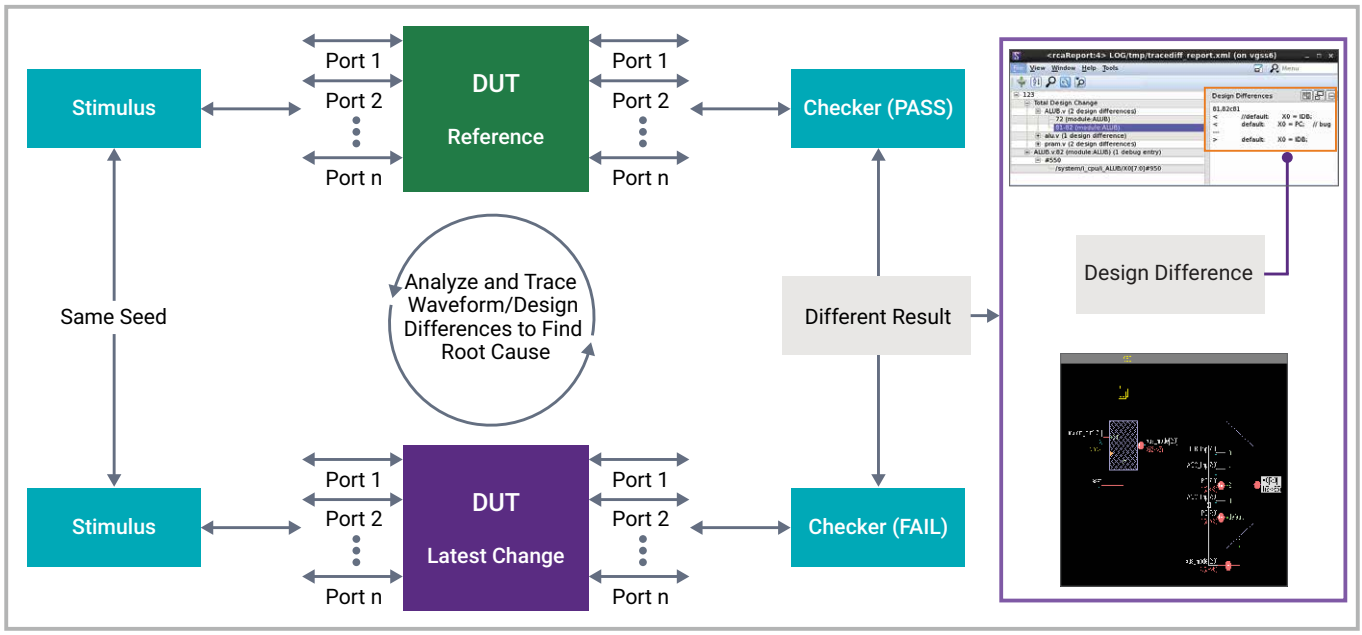

Figure 5: Comparing signal values in DUTRCA

To root cause testbench failures, Debug Facilitator automatically collects debug data for each failure bin. It then facilitates the debug with Protocol Analyzer, showing transactions and associated details along with the reverse debug capability to view the source of the issues back in time. As shown in Figure 6, failing tests are automatically rerun in simulation with reverse debug and other key debug features enabled. Debug Facilitator automatically captures checkpoints during the simulation, essentially the full state of the environment at the time of failure. In addition to that, other important checkpoints are captured. These checkpoints can be used once the RCA results are brought into Verdi interactive mode. Verdi's debug features, including reverse debug, enable quick analysis of the identified failure root causes in the testbench.

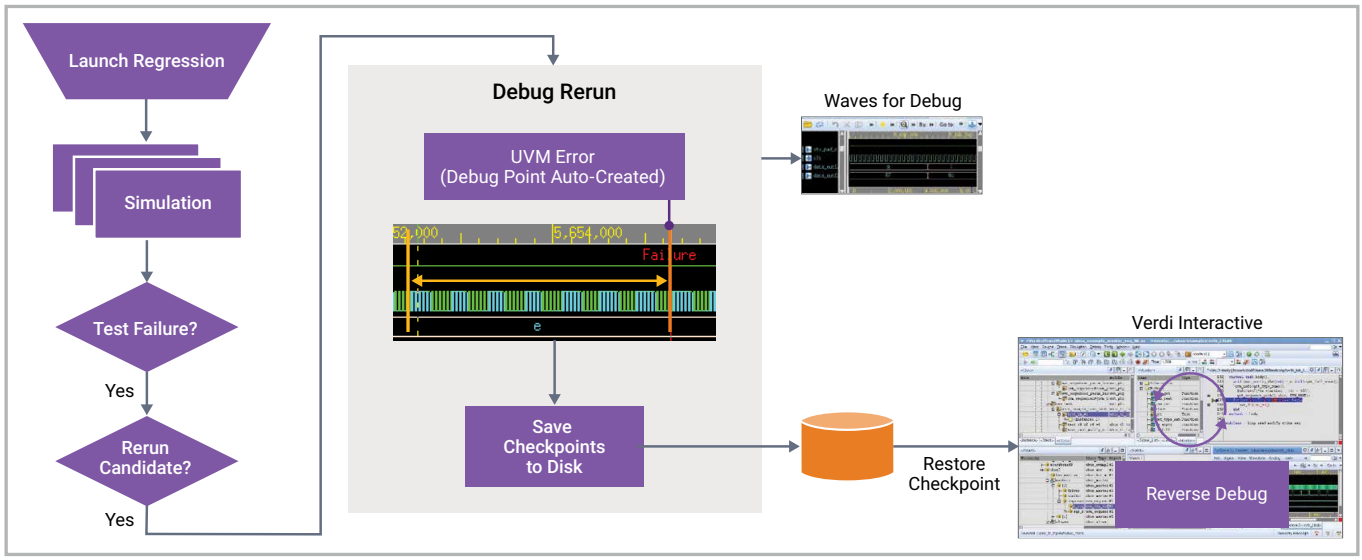

Figure 6: Using checkpoints in Verdi

Verdi RDA includes valuable features to reduce the number of failures related to unknown (X) values that must be analyzed by users. This is important because Xs are notoriously difficult to debug. They typically cover many cycles and levels of logic, and they result in many fanout cones of logic with a single root cause. XRCA is a technology that analyzes single and multiple X paths to the root causes. If this results in multiple paths being sourced from a single root cause, those paths are grouped into a single group. As shown in Figure 7, these results are captured in a report that can be brought into Verdi for analysis and understanding to quickly fix the issues. XRCA automatically scans the X signals in FSDB files and can handle large numbers of such signals to reduce user debug time.

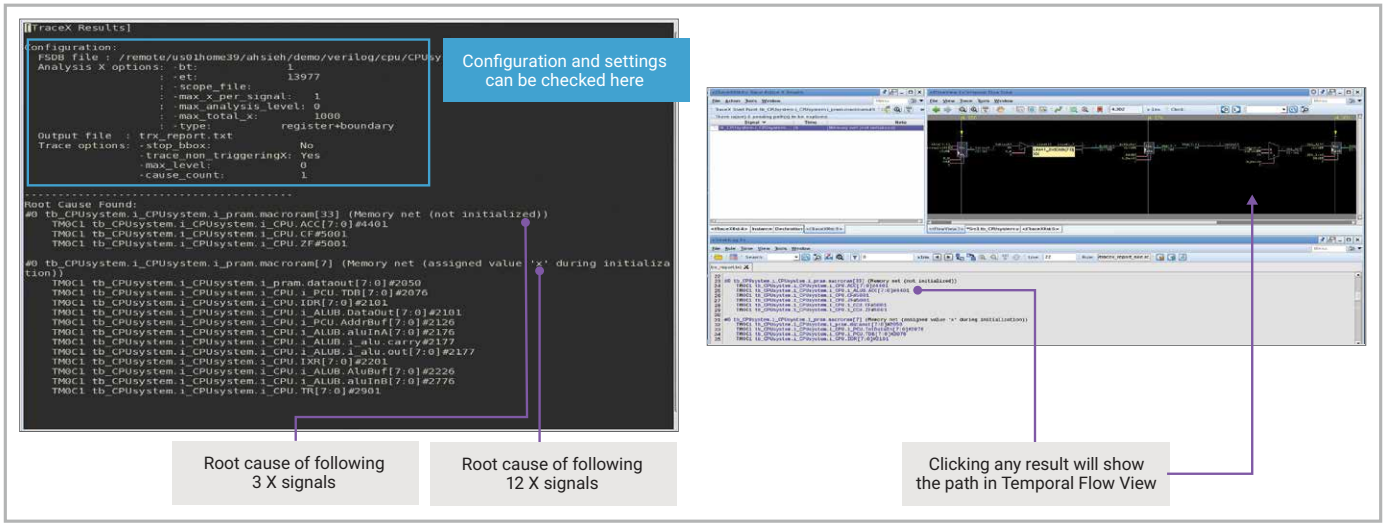

Figure 7: Reducing failures due to unknowns in XRCA

XRCA also addresses X-pessimism, another issue that makes debugging unknown-related failures challenging. Simulation assumes that any unknown inputs to a gate should be propagated to its output. This can be pessimistic, propagating “false” Xs throughout the design. The circuit shown in Figure 8 is a simple example. A signal and its complement are both fed into an “or” gate. If the signal is unknown in simulation, an X value is propagated to the output of the gate. However, since the inputs to the “or” function can only be 01 or 10, its output will always be 1. Eliminating such a failure from the debug effort is highly beneficial. XRCA includes a formal engine that filters out those types of scenarios from the real Xs. It reports both in separate sections so all can be reviewed in Verdi but performs full RCA only on the real Xs to increase the throughput of the analysis.

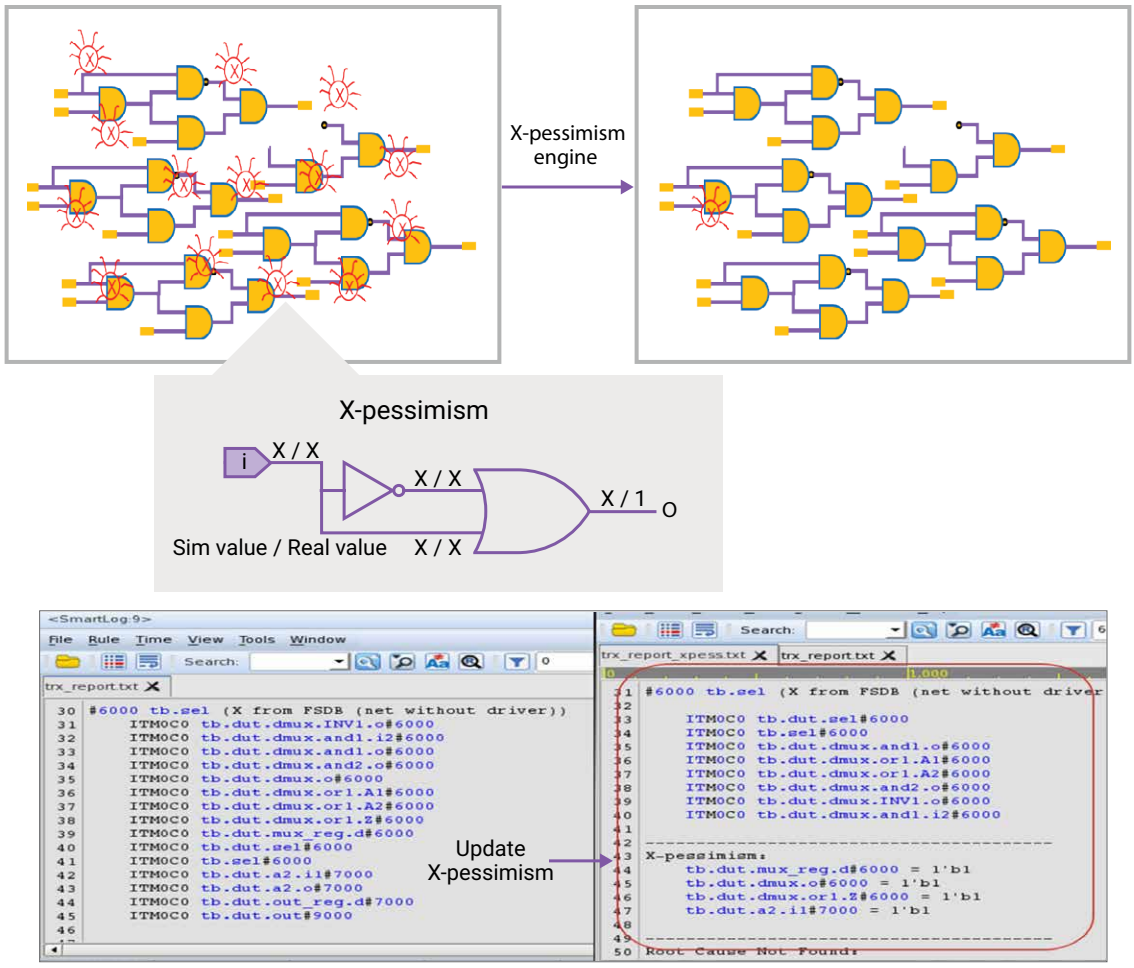

Figure 8: Reducing X-pessimism with XRCA

Summary

Regression Debug Automation in Verdi provides the industry's best solution to the challenges of debugging the results from large simulation regression runs. Automated regression binning requires minimal setup and frees the verification team from the manual task of examining hundreds of thousands of failures. Automated triage separates design and testbench failures and root cause analysis identifies the most likely source of bugs. Debug Facilitator automatically reruns failing tests and provides advanced visualization capabilities to find and fix bugs. Verdi RDA saves significant time and effort for every failing test that is debugged while greatly reducing the number of such tests. The result maximizes regression utilization, focuses manual effort on true debug rather than automatable tasks, reduces the turnaround time (TAT) in the debug process, and cuts the overall debug regression effort on a chip project in half.

08/30/22.CS933416539-SRG-Verdi-RDA-WP.

Pub: July 2022