Failure Triage: The Neglected Debugging Problem

Sean Safarpour, Evean Qin, Yu-Shen Yang
Vennsa Technologies, Inc.
Toronto, Canada
{sean, evean, terry}@vennsa.com

Brian Keng
University of Toronto
Toronto, Canada
briank@eecg.toronto.edu

ABSTRACT
Verification and functional debug are important challenges in the design of integration circuits. With the proportion of time spent on debug increasing, all facets of the problem must be analyzed and improved. *Failure Triage*, which is an important and challenging phase of the debug effort has been mostly neglected by the verification community. Triage is commonly known as the initial phase of debug where a large set of failures are analyzed and grouped together based on the likelihood of sharing the same root cause. Another important aspect of triage is the task of identifying the rightful owners of the problems as quickly and efficiently as possible without wasting other engineer's time. Triage is an important step of verification since it can greatly affect the speed at which bugs get fixed and the general efficiency of the debug process. In this paper, we present a novel failure triage framework that is fully automated and results in much higher accuracy than existing techniques. The framework relies on generating unique signatures for failures based on the suspects derived by root cause analysis engines as well as binning the failure based on clustering algorithms. We use two case studies to illustrate the effectiveness of the proposed framework.

Keywords
Debug, Triage, Regressions, Root cause analysis.

1. INTRODUCTION

Functional verification is a major bottleneck in the design process of integrated circuits. An increasingly large portion of the verification challenge is due to the debug of functional failures. It is estimated that up to 50% of the total verification time can be attributed to debug [1]. In the context of functional verification debug tasks can include determining the root cause of a failure, analyzing the conditions that led to the error, and rectifying the incorrect behavior. With the size of designs increasing, their associated trace lengths getting longer, and verification environments becoming more complex, the debug challenge is becoming more prominent. Furthermore, there is an inherent uncertainty surrounding the debug process in terms of time, engineering resources and difficulty. For instance, it is hard to predict from an error message whether a bug will be fixed in a matter of minutes, or hours, or days.

In the recent past, debug has received some attention from the industry in terms of better tools and methodologies. For example, one of the main benefits of Assertion Based Verification (ABV) is catching failures earlier and thus reducing the time to debug. The main challenge with ABV is that there is seldom enough resources available to implement the desired quantity and quality of assertions. On the tools side, debuggers such as Verdi, DVE, SimVision and Questa have put a big emphasis on accelerating debug tasks, while a new breed of root cause analysis tools such as OnPoint automate some of the manual analysis tasks. On the academic front, there is a constant stream of effort in more effective circuit diagnosis techniques and post-silicon debug [3-4]. While these methodologies, technologies and tools help engineers identify the precise cause of the originating error, the industry as a whole appears to have neglected another face of the debugging problem, that of *Failure Triage*.

Failure triage is often defined as the first phase of debugging in a regression verification flow. It entails identifying the cause of a problem at a higher level than required for root cause analysis and correction of the design. Consider a design where nightly regression tests are run and correctness checking is performed with the aid of multiple checkers, assertions, protocol monitors and Verification Intellectual Property (VIP). When dozens or hundreds of functional failures occur overnight, determining the relationship between these failures is not a trivial task. For instance, how can one quickly determine which of the failures are due to the same bug? If one bug is fixed how many other failures will it also fix? One of the toughest problems is quickly determining which engineer to assign the problem to. Imagine that there are a dozen design engineers and another dozen verification engineers in a group. If the error message is not localized, it is quite difficult to determine which engineer is the best suited person to deal with the problem. Chances are that the problem will be passed from one engineer to another, wasting many people's time, before the rightful owner of the bug is identified.

Today failure triage is performed using one of two widely adopted, yet ineffective approaches. One approach is to dedicate an engineer to triage failures on a daily basis. The task of this engineer is primarily to identify the best suited engineer to further debug the problem. The effectiveness of this approach is based the amount of time the engineer spends analyzing each failure and guided by his inherent knowledge of the system and mostly by his "gut feel". The second approach relies on automation where a simple script groups failures into bins based purely on the error message and the owners of the failing tests. It is clear that the two techniques provide a trade-off between accuracy and speed. The downside of these triage techniques range from inefficient use of engineering resources to the possible incorrect categorization of bugs.

In this paper we investigate the problem of failure triage in detail. First, we summarize the problem and focus on what aspects make it a challenging problem to solve. To address the shortcomings of today's strategies, we introduce a new triage technique using automated root cause analysis tools. The proposed approach is based on the assumption that root cause analysis tools can determine the excitation and propagation paths of the bugs in the design. These paths can act as signatures to differentiate between distinct bug sources. Unlike the signatures produced by checkers, which only provide information about the observation of the bugs, these signatures contain information

about the internal behavior of the design in the presence of the bugs. With some analysis, distinct bug sources can be differentiated and similar ones can be grouped together. The end result of the proposed technique is a set of bins where there is a high likelihood that failures in the same bin are caused by the same bug. Using these bins, bugs can be more confidently assigned to their rightful owners with less time wasted due to incorrect assignments or due to revisiting the same bugs multiple times.

We use two small case studies to illustrate how the proposed flow is applied in practice. We provide experimental data on many different types of bugs caught by different types of checkers and assertions to assess the effectiveness of the approach. It is shown that the proposed approach can be automated in the regression flow and that its results consistently outperform triage techniques based on checker signatures. We conclude the work with in depth discussion on the binning results and their significance.

2. MOTIVATION

A key aspect of failure triage is determining the general area of the bug so that it can be sent to the right engineer. This may appear to be a simple problem but consider this: determining the general location of a bug so it can be sent to its owner requires debugging, but this debugging is probably best performed by the owner himself! In reality, what commonly ends up happening is most failure triage is performed using an error message from failing checkers or assertions. Although this typically provides better than random prediction, it can result in imprecise groupings especially in cases where the number of checkers or assertions are sparse.

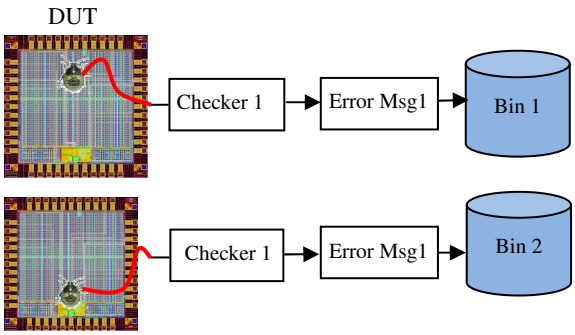

Figure 1: Two distinct bugs are caught by the same checker

For instance, consider a verification regression environment, where hundreds of tests are run every night. Some number of these tests will fail and will be caught by a small set of functional checkers and assertions. Typically, the user will intervene and analyze the failures based on the error messages residing in a simulation log file. The very first question to address is which of these failures originate from the same root causes. A false assumption, that is often made, is that all failures caught by the same checkers are due to the same bug. Indeed, it is possible that many different bugs reside in the design and as the stimulus changes through different tests, different bugs are activated, yet still caught by the same checker. Figure 1 shows this scenario where the different bugs are caught by the same checker and ideally grouped in different bins. The opposite scenario is also common, where due to different stimulus, a bug will propagate through different paths and is eventually caught by different checkers. Figure 2 presents the latter scenario in which,ideally, both failures are grouped in the same bin.

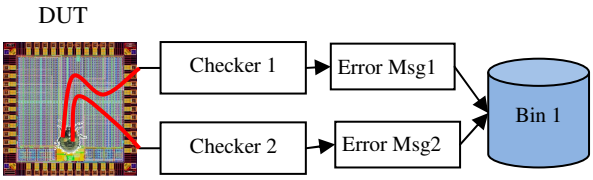

Figure 2: A single bug is caught by different checkers

The lack of information about the bug source is the primary drawback of just using error messages to triage failures. An error message is just one method to generate a *signature,* i.e. method to identify a failure. Error messages give only a small view of the resulting failure. One method to enhance the signature is to consider more messages from the simulation. For example, consider a case where a state machine prints out every state transition it performs. Using the error message along with the state transition can provide the user a better signature based on the state sequences. In general, when more detailed information about the failure is available, such as propagation paths, it is possible to more accurately group similar failures together. In this paper, our aim is to leverage the information available from root cause analysis tools to provide insight into the inner workings of the design.

3. AUTOMATED FAILURE TRIAGE

In this section we present our failure triage methodology with a high-level description of the overall algorithm. This is followed by a presentation of how to generate effective signatures that capture the unique behaviors of different bug sources. Finally, we present our binning algorithm to group similar failures together based on their signatures.

3.1 Overall Flow

Our automated failure triage algorithm helps the triage process by automatically analyzing a set of failures for a given design and grouping them together according to their bug characteristics. This solves several key aspects of the failure triage process. First, it reduces the amount of root cause analysis time required for each failure by grouping similar failures together and separating dissimilar ones. This translates into more distinct failures being analyzed in a given amount of time. Second, it reduces the amount of wasted time due to passing failures back and forth among engineers when the rightful owner is not correctly identified. By sending the failing testcase to the best suited engineer on the first try, significant savings can be gained. Finally, it reduces the possibility that a distinct error goes unanalyzed due to over grouping based on the same failure message.

The overall failure triage flow is shown in Figure 3. It begins with a typical verification flow where many simulation tests are run during nightly regression testing. For each failure, a signature characterizing the error is generated. A rudimentary signature could be as simple as a set of user messages but here an automated root cause analysis tools is more effective. It provides a stronger characterization of the error by implicitly defining possible error excitation and propagation paths. With a signature for each failure, they can then be grouped together into bins where

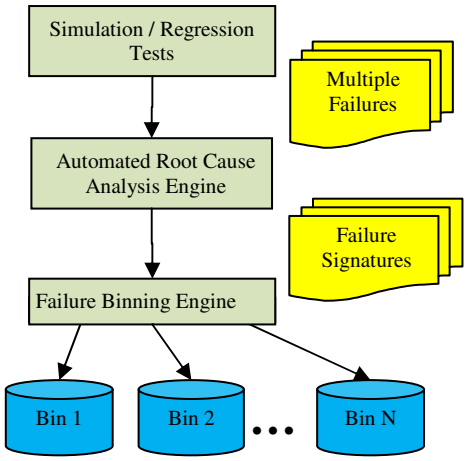

Figure 3: Failure Triage Flow

each bin represents similar failures. With the bins available, the triage engineer can then take the results and send each bin to the appropriate engineer. The signatures (i.e. paths generated) can also provide a hint the next engineer in line about where to start his own debug and rectification process. These steps are described in detail the next sections.

3.2 Signature Generation

As previously discussed, error messages typically do not provide sufficient information to uniquely characterize the failing tests. To illustrate this let's consider two extreme cases. In the first case, let's make the impractical assumption that every signal in the RTL has an assertion or checker verifying its correctness. In the second case only a single checker on the primary output is available. In the first case, the error messages from all the checkers will form a very accurate picture of the failure, detailing precisely the signals where the bug was first excited up until its propagation to the externally observable point (i.e. the final checker or primary outputs). Whereas in the second case, the error message on the primary output only provides a very limited view of what occurred inside of the design. In these scenarios one could easily differentiate between distinct bug sources in the first case versus the second case. Our goal is to generate a signature that can provide information mimicking the former case, rather than the latter, without the unrealistic requirement of adding countless assertions and checkers to the design.

The main problem to overcome is how to find the excitation and propagation paths of bugs automatically in order to generate the signature. Fortunately, automated diagnosis tools [2, 3] or commercial root cause analysis engines such as OnPoint [4] can approximate the excitation and propagation path for each failure. One of the simplest root cause algorithms is critical path tracing [2]. Starting from the failure point, critical path tracing traverses paths backward through the circuit to identify paths and circuit elements that can be sensitized to change the value of the erroneous output. This provides a proxy for the error propagation path. Other more complex algorithms based on formal techniques such SAT, QBF and SMT solvers can also be used to provide even more accurate results [2, 3]. In general one can treat these tools as a blackbox that outputs a set of paths (or circuit elements) given the circuit, an error trace and an expected value as shown in Figure 4.

Commercial root cause analysis tools such as Vennsa's OnPoint can further characterize the failure by providing a set of waveforms that depict what values can rectify the failure at every circuit node or signal. These waveforms also contain an "activation time" that depicts the exact simulation time that the bug is active (or that a fix is required). These paths along with the fix values and activation times provide the user with insight on how the circuit is misbehaving in the erroneous case and how it can be fixed.

In addition to the error propagation paths, more information from the environment can help further characterize bugs. First, user messages, although not sufficient, can provide added information about the test setup. For example, consider a case where the state machine print outs every time it receives a new packet and processes it. The user messages can help build the sequence of events that led to the failure. One approach is to do a coarse grain binning based on the user information first followed by refined binning by the root cause analysis engine. Second, change history from source control software provides valuable information regarding bug locations. For example, if three files have been changed since the last passing regression run, then the owners of these files should be first in line to review the failures.

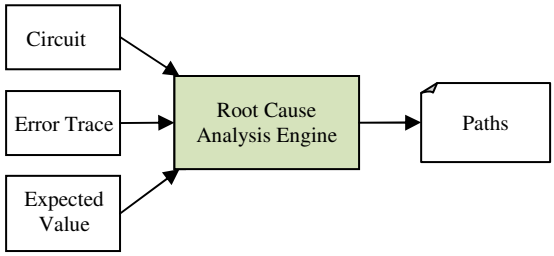

Figure 4: Input and Output of Root Cause Analysis Engine

Lastly, information about the specific stimulus vectors and test sets can be used to help identify similar failures. For example, different modes of a system might imply important clues regarding the locations of errors. All this information can improve the accuracy of the triage process.

3.3 Failure Binning

The process of failure binning takes the failure signatures and groups them together according to their similarity. Although this process can be dependent on the signatures, we show some general binning strategies that can be tweaked for individual environments. We first discuss how to correlate root cause analysis results, then discuss how to use the environmental components of the signature to aid in binning, and finally move on to using these results to make bins.

The key idea when dealing with path based signatures is to re-construct the error propagation paths. By comparing the correlation of two given error paths, the binning algorithm determines whether the sources of the failures are the same. The algorithm uses this information about signals and their activation times to group and sort them in chronological order. Although not always exact, due to cycles in paths or overlapping paths, this ordering provides a proxy for the error path. Next, each path is assigned a score based on the commonality or similarity with

another. If the similarity score is high then two failures should be grouped together; if they are low, then they should be separated.

Other information can be used as additional metrics to either bias the weights when comparing separate error paths, or simply used as tie-breakers when two failures are borderline similar. For example, recently changed code could act as a simple filter to disregard, or change the weight of the different components in the path. Another example is using different operation modes as a tie breaking score for borderline similar failures. These metrics are much more dependent on the environment and can be tuned for optimal use.

Once a similarity score is generated from two signatures, there are many clustering algorithms that can be applied to group them. These algorithms typically involve a threshold parameter that will decide how easily two similar failures can be grouped together. This threshold value need not be static either as it can change based on the environment information as well. In fact, it may be best to experiment with many such variables until settling on an appropriate set of thresholds and metrics.

Finally, when the bins are created, the best suited engineer to fix the problem must be identified. The source control database can tag engineers based on the owner of the most common modules/files or the author of the last change committed for each bin.

4. CASE STUDY

The failure triage infrastructure described in this paper has been developed and is available for commercial use. Its industrial use has been applied to commercial designs in communication applications. Due to the confidential nature of the commercial designs, we cannot disclose the level of data required for this paper. However, to provide a detailed level of information and illustrate the effectiveness of the triage infrastructure, we have collaborated with graduate students from the University of Toronto where the triage tool was applied on two sample designs. To create a realistic verification environment, dozens of "typical" bugs were created in the RTL and testbench components of the designs. A sample of the bugs are shown in the remaining of the paper.

In this section we describe in detail the two case studies. For each we present an overview of the design, provide a sample of the failures found during simulation and show the results of the triage engine.

4.1 Design 1: FPU module

The FPU design used in the case study is a single precision IEEE 754 compliant Float Point Unit (FPU) from Opencores [5] with some modifications. The design is written in Verilog and is composed of eight modules totalling 1415 lines of code. It can perform six operations and supports four rounding modes. The architecture consists of a floating point exception number units, floating point pre-normalization unit that adjust the numbers to equal exponents, primitive operation modules, and floating point post-normalization that denormalizes and rounds the result.

The test suite contains a test set for each FPU operation with different round modes. The test sequences are pre-generated and stored in vector files. Depending on the operation and round mode, the corresponding test sequence is loaded into the memory, as well as the expected values. There is an end-to-end checker that compares the expected value for each operation, exception checkers and some assertions throughout the design.

When simulating the design with the different test sequences we get many failures that occur. Due to space constraints we only show some of the firings as follows.

```
      "14540: ERROR: output mismatch. Expected f292e945, Got
f309efe9 (3ff759808cd7826af292e945) in vector: 4"

      "27540: ERROR: output mismatch. Expected f00007b2, Got
 efcda8a0 (cd7fa2441cff92e8f00007b2) in vector: 17"

      "33540: ERROR: output mismatch. Expected 795a1f75, Got
 79804398 (7b9e426741b9bdbf795a1f75) in vector: 23"

      "34540: ERROR: output mismatch. Expected 35804398, Got
 35dae339 (b3e7a98fbde72f7a35804398) in vector: 24"

      "** Error: Assertion error.
          Time: 1150 ns Started: 950 ns  Scope:
 test.dut.chk_fpu.a_div File: ../sva/fpu.sv Line: 233"

      "ERROR: Underflow Exception Expected: 0, Got 1
       45540: ERROR: output mismatch. Expected 00000000, Got
 00000000 (8a314ad1997a7e9b00000000) in vector: 35"

      "24540: ERROR: output mismatch. Expected ceac709c, Got
 cf2c709c (4ef3129a4f4fc19bceac709c) in vector: 14"

      "49540: ERROR: output mismatch. Expected 45aad895, Got
 462ad895 (c17e453045ab57b845aad895) in vector: 39"

      "** Error: Assertion error.
          Time: 1350 ns Started: 1250 ns  Scope:
 test.dut.u1.chk_pre_norm.a_check_pos_sign File:
 ../sva/pre_norm.sv Line: 70"

      "** Error: Assertion error.
          Time: 2650 ns Started: 2550 ns  Scope:
 test.dut.u1.chk_pre_norm.a_check_neg_sign File:
 ../sva/pre_norm.sv Line: 75"

      "43540: ERROR: output mismatch. Expected 6fcfb17a, Got
 efcfb179 (6fcfb17a1bb73bd36fcfb17a) in vector: 33"

      "48540: ERROR: output mismatch. Expected aebaa9dd, Got
 2ebaa9dd (aebaa9de996ed347aebaa9dd) in vector: 38"

      "ERROR: DIV_BY_ZERO Exception: Expected: 1, Got 0
       28540: ERROR: output mismatch. Expected 00000000, Got
 00000000 (92bf785f9b6e56a400000000) in vector: 18"
```

Notice that some errors are due to assertion failures and others are due to golden value mismatches and exception catching. We run the OnPoint root cause analysis engine and use the result to generate the signatures described. The clustering algorithm groups all the failures into five bins. After performing further manual debugging on each bin, the root cause of each error is identified. In these experiments, the grouping is performed correctly as the same bugs are grouped together and distinct bugs are grouped separately. Note that if binning was done purely based on the failure messages there would have been four bins where at least two bug sources would have gone unidentified, and another bin would have presented a duplicate error.

```
Bin 1: 4 checkers, 1 assertion

    Bug Location: primitives.v : 90 & 98
    ->  // Bug: missing one clock delay
    ->  always @(posedge clk)
    ->      quo <= #1 opa / opb;
    ->  always @(posedge clk)
    ->      rem <= #1 opa % opb;

    ->  // Fix:
    ->  always @(posedge clk) begin
    ->      quo1 <= #1 opa / opb;
    ->      quo  <= #1 quo1;
    ->  end
    ->  always @(posedge clk) begin
    ->      rem1 <= #1 opa % opb;
    ->      rem  <= #1 rem1;
    ->  end
```

Next, each of the bins containing the root causes are briefly described. The first bin groups four checker failures and one assertion failure together, which is typically hard to do manually.

The correct and buggy RTL is shown above. In this case the bug is that there is a missing pipeline stage.

```
Bin 2: 1 exception

    Bug Location: test_top.v : 322 - 326
    ->  // Bug: incorrect stimulus
    ->  ...
    ->  @(posedge clk);
    ->  #1;
    ->  ...
    ->  oper  = tmp[103:96];
    ->  ...
    ->  case(oper)
    ->      8'b00000001:      fpu_op=3'b000;      // Add
    ->      ...
    ->      8'b01000000:      fpu_op=3'b110;      // rem
    ->      default: fpu_op=3'bx;
    ->  endcase
    ->  ...
```

The second bin contains a single exception error. The bug in this case resides in the Verilog testbench where bad stimulus is generated. Interestingly this failure is distinguished from the others and is binned on its own.

```
Bin 3: 2 checkers

    Bug Location: post_norm.v : 354
    ->  // Bug: Incorrect padding bit
    ->  assign {exp_rnd_adj0, fract_out_rnd0} = round ?
fract_out_pl1 : {1'b1, fract_out};

    ->  // Fix:
    ->  assign {exp_rnd_adj0, fract_out_rnd0} = round ?
fract_out_pl1 : {1'b0, fract_out};
```

The third bin groups two checker failures. In this case a basic grouping algorithm would have resulted in a similar result. This bug is in the RTL and is due to setting the top-most bit to zero instead of one.

```
Bin 4: 2 assertions, 1 checker, 1 exception

    Bug Loaction: pre_norm.v : 213 - 216
    ->  always @(signa or signb or add ...
    ->          ...
    ->          // Bug: switched assignments
    ->          3'b0_0_0: sign_d = 1;
    ->          3'b0_1_0: sign_d = !fractb_lt_fracta;
    ->          3'b1_0_0: sign_d = fractb_lt_fracta;
    ->          3'b1_1_0: sign_d = 0;

    ->          // Fix:
    ->          3'b0_0_0: sign_d = fractb_lt_fracta;
    ->          3'b0_1_0: sign_d = 0;
    ->          3'b1_0_0: sign_d = 1;
    ->          3'b1_1_0: sign_d = !fractb_lt_fracta;
```

Bin four groups two assertion failures and two checker failures, which is typically hard to identify manually. The bug is in the RTL and is a result of incorrectly decoded signals inside a case statement.

```
Bin 5: 1 exception

    Bug Location: test_top.v : 302 - 306
    ->  // Bug: incorrect reference model
    ->  if(div_by_zero != exc4[2])
    ->      begin
    ->          exc_err=1;
    ->          $display("\nERROR: DIV_BY_ZERO Exception:
Expected: %h, Got %h\n",exc4[2],div_by_zero);
    ->      end
```

Bin five captures a single exception on its own. In this case, after root cause analysis, it finds that the bug is in the expected model used for the exception handling. This strengths the notion that the triage approach is also valid for bugs outside of the DUT.

4.2 Design 2: VGA controller

The VGA controller is from Opencores [5] with some modification, it is written in Verilog, composed of 17 modules totalling 4,076 lines of code and approximately 90,000 synthesized gates. The controller provides VGA capabilities for embedded systems. The architecture consists of a Color Processing module and a Color Lookup Table (CLUT), a Cursor Processing module, a Line FIFO that controls the data stream to the display, a Video Timing Generator, and Wishbone master and slave interfaces to communicate with all external memory and the host, respectively.

The operation of the core is as follows. Image data is fetched automatically via the Wishbone Master interface from the video memory located outside the primary core. The Color Processor then decodes the image data and passes it to the Line FIFO to transmit to the display. The Cursor Processor controls the location and image of the cursor processor on the display. The Video Timing Generator module generates synchronization pulses and interrupt signals for the host.

```
        "# ** Error: Assertion error.
         #    Time: 603 ns Started: 603 ns  Scope:
test.dut.chk_top_i1.assertion_a_fifo_rreq File:
../sva/vga_top.sv Line: 92"

        "# ** Error: Assertion error.
         #    Time: 609 ns Started: 603 ns  Scope:
test.dut.wbm.clut_sw_fifo.chk_fifo_i1._a_read_pointer File:
../sva/vga_fifo.sv Line: 32"

        "# ** Error: Assertion error.
         #    Time: 609 ns Started: 603 ns  Scope:
test.dut.wbm.data_fifo.chk_fifo_i1._a_word_down_counter File:
../sva/vga_fifo.sv Line: 72"

        "# ** Error: Assertion error.
         #    Time: 897 ns Started: 897 ns  Scope:
test.dut.sigMap_i1.assertion_wbs_dat_o File:
../sva/VennsaChecker.sv Line: 45
         #     897.0 ns: expected aaaaaaaa, got ffffff9f"

        "# ** Error: Assertion error.
         #    Time: 633 ns Started: 627 ns  Scope:
test.dut.wbm.data_fifo.chk_fifo_i1._a_read_pointer File:
../sva/vga_fifo.sv Line: 32"

        "# ** Error: Assertion error.
         #    Time: 651 ns Started: 645 ns  Scope:
test.dut.pixel_generator.rgb_fifo.chk_fifo_i1._a_read_pointer
File: ../sva/vga_fifo.sv Line: 32"

        "# ** Error: Assertion error.
         #    Time: 831 ns Started: 831 ns  Scope:
test.dut.sigMap_i1.assertion_wbs_dat_o File:
../sva/VennsaChecker.sv Line: 45
         #     831.0 ns: expected ffffff9f, got ffffffXf"

        "# At time     273.0 ns: ERROR in wishbone:
golden=0000000000000000, actual=0000000100000000"

        "# ** Error: Assertion error.
         #    Time: 273 ns Started: 273 ns  Scope:
test.dut.sigMap_i1.assertion_wbs_dat_o File:
../sva/VennsaChecker.sv Line: 45
         #     273.0 ns: expected 00000000, got 00000001"

        "# At time     111.0 ns: ERROR in sync: golden=0,
actual=1"
```

The test suite for the VGA core is constructed using UVM. Four main tests are used for verifying this design. These include register, timing, pixel data, and FIFO tests. The transaction has randomly generated control-data pairing packets under certain constraints. These transactions are expected to cover all the VGA operation modes in the tests (and they may be reused to test other video cores such as DVI, etc). The sequencer exercises different combinations of these transactions through a given testing scheme so that most corner cases and/or mode switching are covered. The monitors are connected to the DUT and the reference model

respectively. They check the protocols of the responses, and make sure that the data being sent to scoreboard has correct timing. The scoreboard and checkers contain all the field checkers which compare the data from the DUT, and reports the mismatches.

The golden reference model is implemented using C++. It receives the same set of stimulus from the driver (uvm_driver class) and produces the expected value of the outputs. Along with the reference model, 50 SystemVerilog Assertions (SVA) are used to do some instant checks. While running simulation, SVA can catch unexpected behaviours of the design and prevent corrupted data going through the flow.

A sample of the failures that occurred during a suite of simulation tests is shown. Notice that there are a set of assertions and correctness checkers that fire. As in the FPU case, OnPoint is run on the test to generate the suspects which are used as signatures during the triage process.

If triage were performed based purely on the error message the result would be six bins. In contrast there are only four errors in this case, thus time would have been wasted analyzing redundant failures. Furthermore, two of the errors could also have been missed if only one failure is analyzed within each bin. In contrast, the triage infrastructure proposed correctly generates four bins, one for each error. The resulting triage bins and the root cause of the failures are shown below.

```
Bin 1: 4 assertions (3 different)

    Bug Location: vga_colproc.v : 263
    ->  always @(c_state or vdat_buffer_empty or colcnt or
DataBuffer or rgb_fifo_full or clut_ack or clut_q or Ba or Ga
or Ra)
    ->      begin : output_decoder
    ->
    ->              // initial values
    ->          // Bug incorrect initial value
    ->              ivdat_buf_rreq = 1'b1;
    ->
    ->          // Fix:
    ->          ivdat_bug_rreq = 1'b0;
```

Bin one groups four assertions failures based on three different assertions thus eliminating wasted time by analyzing each one separately. The single bug source is due to an incorrect assignment based on the state of the vga color processor.

```
Bin 2: 3 assertions (3 different)

    Bug Location : vga_fifo.v : 191
    ->    always @(posedge clk or negedge aclr)
    ->     ...
    ->      // Bug: missing use of function
    ->        else if (frreq) rp <= #1 {rp[aw-1:1], rp};
    ->      // Fix:
    ->      else if (frreq) rp <= #1 {rp[aw-1:1], lsb(rp)};
```

Bin two catches three distinct assertion failures once again. In this case, the RTL bug is due to picking the wrong bit of a read pointer inside a fifo.

```
Bin 3: 1 checker, 1 assertion

    Bug Location: test_bench_top.v : 684
    ->  // Bug: incorrect stimulus "wb_err_i" generated from
wb_slv model
    ->  wb_slv #(24) s0(.clk( clk                 ),
    ->    .rst(              rst                ),
    ->    .adr(     {1'b0, wb_addr_o[30:0]}     ),
    ->    ...
    ->    .err(              wb_err_i  ),
    ->    .rty(                                 )
    ->    );
```

Bin three contains both a checker and an assertion failure. Interestingly, this bug resides in the testbench where some stimulus signals are instantiated using the wrong models. As a result both the checker and an assertion fail. Debug such cases typically would involved multiple designers and verification engineers.

```
Bin 4: 1 checker

    Bug Location : self_checking.v : 110 - 121
    ->  // Bug: incorrect "blanc_golden" generated from
erroreous reference model
    ->  if(^{hsync_golden, vsync_golden, csync_golden,
blanc_golden} !==1'bx)
    ->        if({hsync_golden, vsync_golden, csync_golden,
blanc_golden} != {hsync, vsync, csync, blanc})
    ->        begin
    ->            $display("At time %t: ERROR in sync:
golden=%h, actual=%h", $time,
    ->                          {hsync_golden, vsync_golden,
csync_golden, blanc_golden},
    ->                          {hsync, vsync, csync, blanc}
    ->                          );
    ->            ->ERROR;
    ->
    ->        end
```

Bin four contains a single checker failure. This failure is caused by a bug in the testbench where the reference model contains the bug.

In all these cases, we have confirmed that the bins generated by the proposed triage approach correctly bin the failure based on the same root cause. We confirmed the finding by verifying that fixing the bugs remove all the failures for a given bin. It should be noted that the proposed triage approach may not always be correct, if distinct bugs are close in proximity they may end up in the same bin.

5. Conclusion

In this work we presented a novel failure triage approach that is both automated and generates better results than previous script-based and manual techniques. The triage engine relies on information from root cause analysis tools that provide visibility into the propagation paths of the bug. These paths along with their activation times provide unique insight that is used to group similar failure together. To illustrate the effectiveness of the approach we provide two small case studies where distinct bugs are correctly binned separately. Further research in this area will focus on improving the resolution and quality of the binning algorithms and generating custom heuristics for testbench and environment originating bugs.

6. REFERENCES

[1] H. Foster, “Assertion-based verification: Industry myths to realities (invited tutorial),” in *Computer Aided Verification*, 2008, pp. 5–10.

[2] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[3] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, “Fault Diagnosis and Logic Debugging Using Boolean Satisfiability,” *IEEE Trans. on CAD*,vol. 24, no. 10, pp. 1606–1621, 2005.

[4] Vennsa Technologies Inc., http://www.vennsa.com/product_simulation.html

[5] OpenCores, http://www.opencores.org