needed to debug deep within the design, or testbench, or firmware, and one way to achieve that is to use triage to funnel groups of failure analysis tasks to those resources wisely, to achieve the most overall productivity.

A recent job advert for a triage role with a certain fruit themed tech company in California cites the following desired attributes for this job role, and although it refers perhaps more to system-level testing triage than regression-level, it is a sign of this further evolution in design/verification/debug resourcing:

- In your job as triage engineer, your goal is to quickly and effectively process incoming units that failed some software test where the failure is believed to be in hardware…
- You will use diagnostic software tools from content experts. You will enhance and maintain the triage methodology flowchart and associated automation scripts. You will have opportunities to work with experts from many teams to understand SoC and system behavior…
- Your success will be defined by the successful triage of incoming failed units, your triage throughput, the relationships you build, and the improvements you bring to the process.

An engineer performing the triage role will typically have several interactions as shown in Fig. 3: incoming work, informative collateral, process and environment knowledge, infrastructure knowledge.

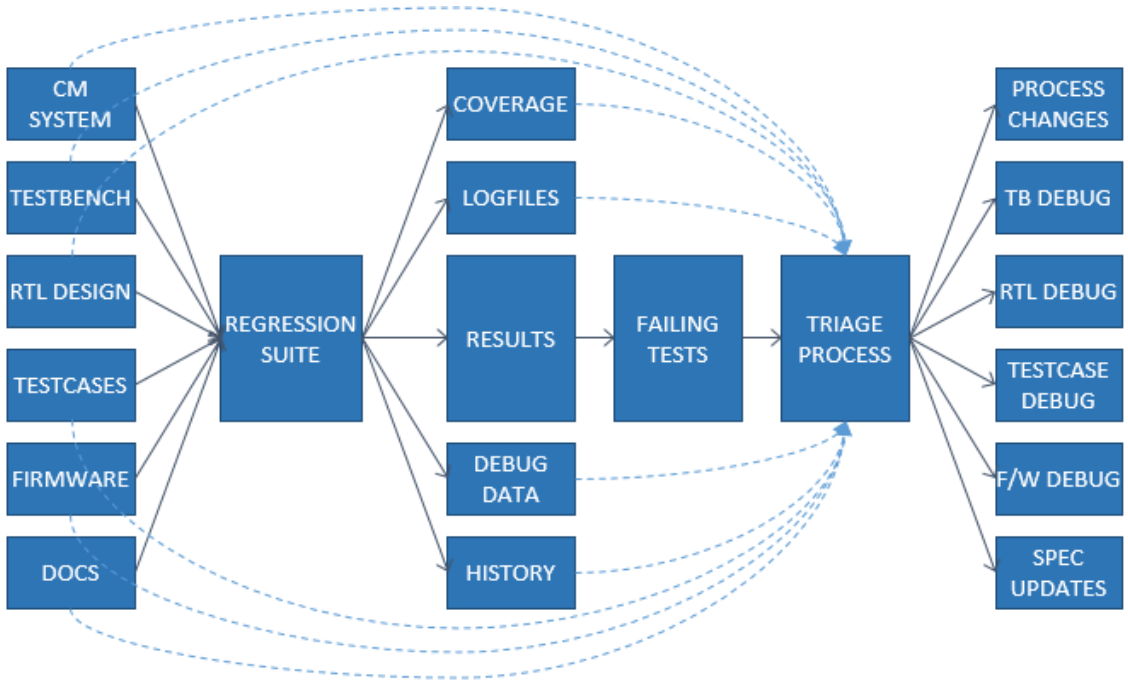

Figure 3. Interactions in the triage process

The incoming work is usually in the form of an automated regression system (the infrastructure) making reports of items to be debugged. The collateral will include requirements specifications or user manuals of the design under test, and the source code and design intent of the testbench and of the design it is testing. Process and environment knowledge comes partly from project managers, partly from CAD departments. The triage engineer may interact with specialists in all these areas regularly.

IV. KINDS OF TRIAGE

As we saw from the classic fields of endeavor where triage originated (medical, military, firefighting), triage is a multi-faceted activity which depends on cumulative experience as much as it does on quick analysis and decision making.

However triage of regression failures in microelectronic design verification has one important difference to triage of medical issues in an emergency or disaster scenario. Those scenarios involve random one-off occurrences of issues in random people, there is (usually) no concept of repetition, or grouping, than can inform the triage process – each case is treated – quickly – in isolation to rapidly categorize the issues into a set of bins: deceased / immediate care / delayed care / minimal care. In our world, grouping and history is important. We can identify several different kinds of triage that occur in design debug, verification and validation flows:

A. Triage of a single failing test to disposition corrective action

At the simplest level, triage is where experience is applied to the assessment of a single failing test case and the decision on how to disposition the problem for further action (deeper analysis / debug or fix / workaround or process improvements. In the SoC context, this may be a simple split between a probable RTL/GLS design bug, or a test environment bug (i.e. a false negative), or a firmware/software related bug, if there is any in the design, or a specification error or ambiguity.

B. Triage of a failing test or failure mode in historical context, to disposition longer term corrective action

A deeper look at the failing test case or firing assertion, looking back across history to answer questions such as 'when did this last fail' or 'why does it continue to fail from time to time' may uncover areas of the design that are susceptible to errors being introduced over time, or areas of the test environment that are overly brittle or fragile in their approach to checking correct behavior. Triage may both disposition this bug fix plus a longer term recommendation, and often only the triage engineer will be in a position to spot that situation

C. Triage of multiple failing tests to determine resource focus and priority

This is the most common application of triage as opposed to fail by fail debug, in a chip design/verification project once it gets into the regression stage where enough of the chip and its testbench exist and enough runnable test cases exist, to make a viable regression suite drive the day to day / week to week activities of the team. A good test environment will add value to the failure analysis process and specifically to the triage / root cause prediction process, by recording meaningful observations and data in a way that the triage engineer can observe commonality of symptom, and possible commonality of root cause. Ultimately the desired resolution of a regression fail is not to treat the symptoms, but to treat the root cause. Triage can provide a head start on that debug activity when there are many failing tests, and we discuss some techniques here.

D. Triage identifying systemic failures

A useful side effect of having a good triage flow at the lower level, is the identification of common failures or kinds of failure, leading to the deduction of common root causes which introduce weakness or instability into the design under test. This can be a result based on good historical data capture with subsequent follow-up correlating root cause data with symptom data. This kind of triage is related to the previous category, but the goal here is not to resolve multiple failing tests at once, it is to look at the bigger picture and capture data which may enable process improvements, or verification environment architectural advances, project after project.

VI. TRIAGE TOOLS

Data gathered from the industry suggests that engineering teams spend 40% of verification time doing debug. The initial part of each debug exercise is triage. It may be performed by the same person as the verification, deeper debug, or even design engineering activities, or by a specialist, depending on team size, so it is difficult to quantify the time spent on triage as a separate activity. This author estimates it comprises the first 20-50% of every debug effort, and is worthy of exploration with a view to making productivity improvements in the debug area as a whole.

When this situation arises in the microelectronic design industry, an EDA solution normally can be found, as it is worthwhile to spend money to improve productivity and hence time to tapeout, time to correct silicon, time to market. What would it take to provide a tool that performs the triage activity for us, though?

Looking at the kinds of triage, we can see limited opportunities for automation:

Type A: triage of a single failing test – this could be analyzed and dispositioned if the failure symptom was a clear indicator, such as an assertion firing at the point of failure. However many fails are symptoms which appear much later or in an unrelated area of the design. Some initial manual tracing or correlation among multiple data sources is required.

Type B, looking at the history of a testcase, can be assisted by some automation, but what deductions could be made by a tool alone, without knowledge of context and intent? At least automation can provide the reporting or exploration mechanism, but not the conclusion.

Type C, looking across multiple fails for common symptoms, is an activity that can be partially automated, and there are various Results Analysis tools and techniques available in the industry which assist in this area, but the majority of the triage activity is taking that information and analyzing further.

Type D, identifying systemic failures over time, can be assisted, and tool automation in this space can come into its own when larger amounts of data are involved – so called "big data" solutions which are good at spotting anomalies, outliers, trends and patterns. However, debug of a regression run is really a "here and now" kind of problem, so while creative automation is possible, it is not a complete solution.

We conclude that tools that implement automation for the four kinds of triage activity listed above do not yet exist. There are tools which can assist the process, and we will describe some algorithms and flows here, but at the current time, triage is a manual activity: the tools can't do it for us.

Reassuringly, this is the case in the medical and emergency services professions also – while processes and flow charts can be developed which help to guide the triage process, it ultimately takes human instincts and professional experience to develop the ability to consistently and fairly triage incoming situations.

V. TRIAGE TECHNIQUES

Triage as a discipline can be learned, benefitting from industry expertise. The following techniques and recommendations were gathered over a number of SoC design and verification projects with teams of various profiles, sizes and compositions. Some common triage techniques were observed and used:

A. Categorize an individual failure

Several techniques can be used here to look at a fail in isolation and make observations about it which will help with other collective failure analysis activity:

1) *When analyzing a regression fail, categorize it by its failure mode. Was it a build-time error or a run-time error? Did it fail due to a testbench-caught fail condition, or a testbench timeout, or a testbench lack-of-success condition, or a golden-reference comparison mismatch?*

2) *Further categorize all the testbench-caught fail conditions: find the first failing error message in the log file. Find warnings that were uncharacteristic of the testcase that may be early indicators of impending failure. Ideally the earliest assertion that fired should be known, as it is most likely to point the way to the root cause.*

3) *Identify the nature of the check that caused the fail: is it a designer inserted assertion, or a verification environment assertion, or a scoreboard check, or a protocol check? What is the purpose of the check and how precise is its diagnosis. In some cases there is nothing more to do, in others there is ambiguity and triage needs to dig deeper or look at related symptoms to build a picture of where to debug next.*

4) *Further categorize any golden-reference comparison mismatch failures: analyze them by time, and by which signal or element mismatched, and record precisely the details as they may be useful for comparison in other parts of the triage exercise.*

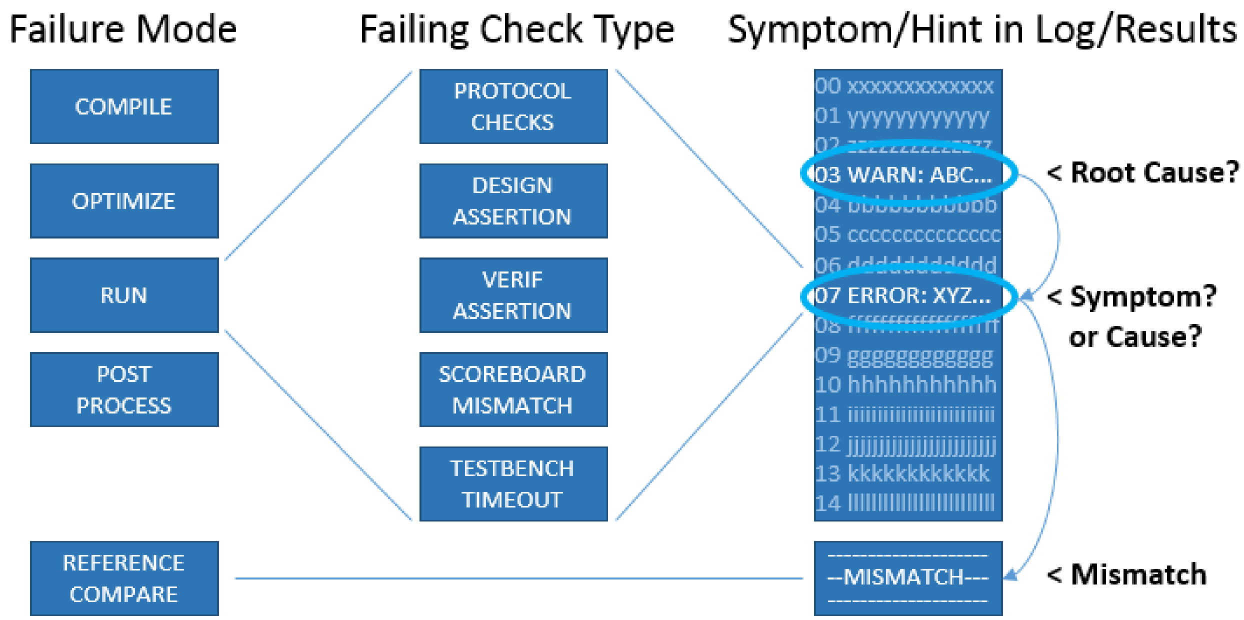

Figure 4. Categorizing a regression failure

The main benefit of this initial preparation in triage is to know which resource or expertise is required and who should dig deeper on this particular fail.

B. Analyze a list of N fails, looking for commonality across multiple fails

The above techniques alone though can lead to wasted effort. If there are multiple fails, then triage the collection of fails as well as each individual fails, to spot trends and groups. Build or buy a tool or scripting environment that can provide analysis of multiple runs and execute database-query-like analysis on the data:

1) *Explore whether the same error message appears across multiple failing test cases or configurations. The error message in question would be the first one that is logged – subsequent errors are by definition further removed from the root cause, so normally only the first error is used. It is desirable that error messages are well structured in order to make programmatic analysis possible. Methodologies such as UVM can assist with providing the required structure [3].*

   For example, you may not be comparing exactly the same error message verbatim, but tolerating a degree of delta especially if the message refers to random data elements which are incidental to the facts of the failure mode. In this case, some filtering is require and the technical requirements for the comparison algorithm can become quite complex. A good Results Analysis solution will accommodate flexible filtered comparison.

2) *If the fail comes from an HDL assertion firing, categorize assertion behavior across multiple tests. In the simplest form this is the same as (1) above if the assertion causes a well-structured error message to be issued. Your simulator may record the assertion in a particular way that provides access to additional data than the user-written error message. Note that well-written assertions will notify of a problem close to the root cause. This is one advantage of assertions over procedural scoreboard code when performing protocol checks, both the signal location and the time region of the errant behavior are precisely located. Did the same assertion fire in multiple runs? Under what variety of circumstances did it fire?*

3) *Sometimes there is less precision but the failure mode may narrow down the incorrect behavior to a general area of design functionality or a group of problematic signals. Look for anything in common across multiple failing tests, so that they can be binned together and inform the next steps in the debug process.*

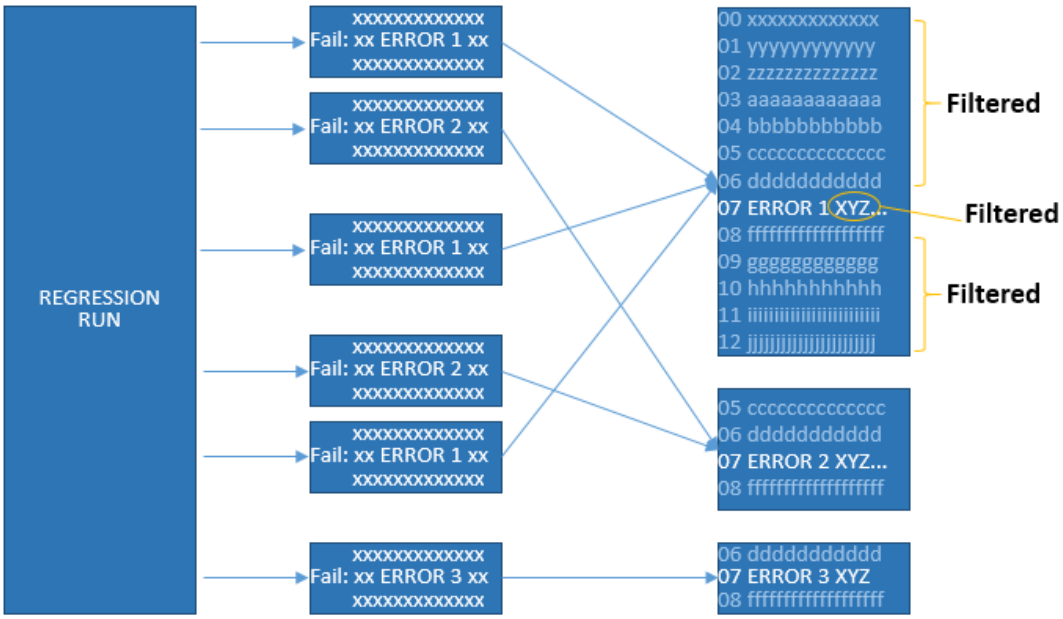

Figure 5. Analysis of N related failure symptoms

Once this kind of grouping is done, an instance from each bin can be triaged using techniques (A) above to further prepare for debug.

C. Analyze a list of NN fails looking even deeper for common potential root causes across multiple fails

Simply analyzing regression log files or assertions fired certainly helps group failing test cases with like symptoms for further debug, but as we have seen, the symptoms of a failing test may be far removed in time or scope from the root cause of that failure. And we may see many failing tests in a regression each with a differing failure symptom, when constrained random stimulus is involved.

We speculate that further grouping can be done once a common root cause is identified, to narrow down the debug work still further. After all, the new failures were normally caused by one of a small number of design or testbench changes since the prior passing regression run. The ultimate triage productivity benefit is isolating the single biggest root cause, affecting many regression test runs as a consequence.

1) *One possible technique is to use formal technology to analyze the design around the point of failure. Tools are available which can automatically apply formal property checking to an identified region of a design to identify possible or impossible paths and spot common design rule violations.*

   The exercise would then be to gather the reports of the formal tool and analyze multiple instances for a common theme. In this way a single design flaw that manifested itself in differing ways depending on test case can be identified.

2) *The root cause analysis features of your debug tool may also be useful here. It is possible to run a causality check in the debug tool given just two pieces of information: a signal name, and a simulation time, on which that signal has a transition that triggered the failing assertion or check (an 'X' check, for example). The causality check analyzes the logic fan-in cone to that signal, looking back through both combinational and sequential logic, back in time using the captured signal values, and deduces a root cause signal and value.*

   The purpose of doing this is to repeat the analysis of 'common symptom signature' that we performed as in 'B' above, but with 'common root cause signature' as a better goal. In a regression run, several different symptoms could arise from one root cause.

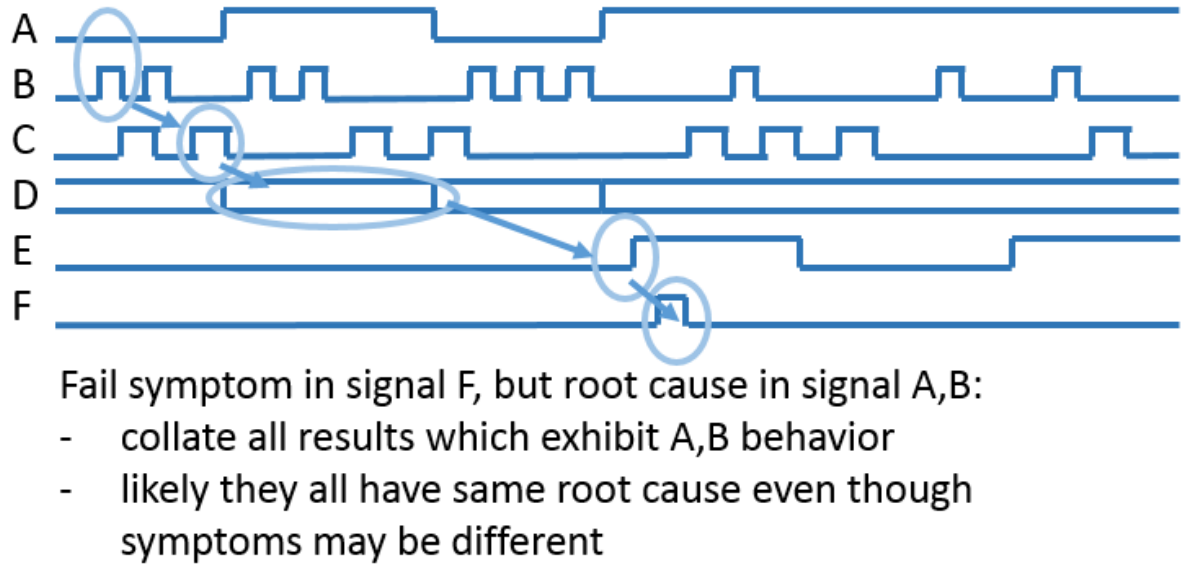

Figure 6. Deeper analysis of N related failure root causes

Identifying these groupings is low hanging fruit for productive debug as it could take care of a large number of failing test cases in one go – even if they happened to have diverse symptoms.

D. Analyze a single test instance across N regression runs of run history

Another way to look at regression data is to look back in time given a common reference point – normally a directed or directed-random test case run with a particular seed and configuration.

In this case the purpose is to derive a temporal grouping of test status – determine the 'normal' status for a test and indicate maturity, or brittleness, of both the test and the design.

This adds tremendous value to the triage process because it can answer several useful questions:

1) *Has a failure mode occurred before? In fact this question need not be confined to one test case or stimulus/configuration combination. It is useful to gather information on when this observation was last made and what was deduced about the problem on that occasion. For this testcase, is this a new regression in behavior, or is the testcase hitting a limit of testbench fragility and now needs to be relaxed to accommodate design behavior that is within specification.*

2) *Does this test fail continuously/regularly or sporadically – is it a Noisy Test? This is useful to know partly in case of an overly brittle testbench and partly to learn why designers or testbench writers might frequently violate this particular check, in order to offer advice on avoiding this failure mode in the future.*

3) *When it last failed, what did the debug session look like, and do we already know the likely root cause? Keeping records of debug sessions to inform future triage is prudent. The initial steps of debug that proved to be valuable last time in finding the root cause, can be reapplied this time when it looks like a reoccurrence.*

4) *Who has investigated this kind of failure before, can we find them and ask / allocate this fail to them? There is no substitute for engineer experience with debug of a particular kind of issue. If it is the outcome of triage that debug actions on failing testcases are allocated among the engineering team, then historical data will help find the right person first time.*

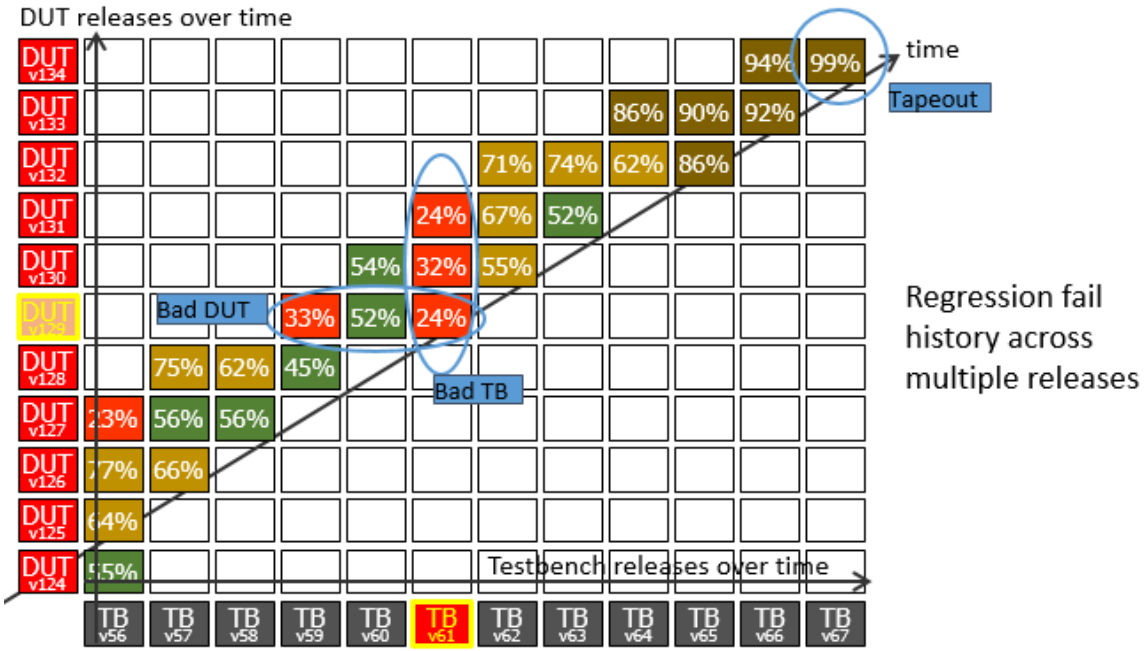

Figure 7. Analysis of regression history for one failure node or many

This is the end goal of the triage phase of debug – to allocate a failing testcase to an expert for further debug and corrective action. Several factors will come into the decision on what to allocate, and who to allocate it to, all based on the information gained by the triage engineer: an evaluation of type of issue (i.e. what expertise will be required next), and historical factors (requiring experience in debug), and severity to the project (identifying the value of the subsequent debug exercise). The goal is to maximize the value of debug, by careful choice of which testcase debug will achieve maximum impact steering the design back on course until the next regression run.

VIII. CONCLUSIONS

Triage is a specialized form of debug activity which can consume significant time and resources, and there is opportunity for optimization of this role and the processes and techniques used to increase overall productivity and avoid the wasted effort of blind debug alleys.

Tools cannot perform the triage role for us per se, but they can help with data gathering, reporting and initial analysis, with the remainder of the analysis being manual based on our experience. Part of the effort is having good data at our fingertips during regression analysis and results analysis phases, including a historical context. Next is to develop a precision in the way decision and factors of interest are communicated to the next person in line to continue with deeper debug. Several techniques are described here which may help these activities.

We leave you with some of the text of the aforementioned Triage Specialist job description:

- Do you love debugging issues that may lie in system software, chips, or boards? Are you disciplined, methodical and reliable? Do you have skills that span CPU and SoC architecture, boards, OS and firmware interactions? Do you have experience with silicon bring-up, validation and debug? Come talk to us.
- We are looking for star performers...

It is hoped that the approach described here helps equip you for enhanced productivity, accuracy and success in your triage and debug flow.

REFERENCES

[1] Merriam-Webster, *English Dictionary*, http://www.merriam-webster.com/dictionary, August 2015.
[2] Robert Klinoff, *Introduction to Fire Protection*, New York, Cengage Learning, 2007.
[3] Tom Fitzpatrick, *Advanced UVM Debugging*, Mentor Graphics, 2013