

Simulation And Satisfiability Guided Counter-example Triage For RTL Design Debugging

Zissis Poulos¹, Yu-Shen Yang² Andreas Veneris¹ Bao Le¹

¹Dept. of ECE, University of Toronto, Toronto, Canada. {zpoulos, veneris, lebao}@eecg.toronto.edu

²Advanced Micro Devices Inc., Toronto, Canada, Yu-shen.Yang@amd.com

Abstract—Regression verification flows in modern integrated circuit development environments expose a plethora of counter-examples during simulation. Sorting these counter-examples today is a tedious and time-consuming process. High level design debugging aims to triage these counter-examples into groups that will be assigned to the appropriate verification and/or design engineers for detailed root cause analysis. In this work, we present an automated triage process that leverages knowledge extracted from simulation and SAT-based debugging. We introduce novel metrics that correlate counter-examples based on the likelihood of sharing the same root cause. Triage is formulated as a pattern recognition problem and solved by hierarchical clustering techniques to generate groups of related counter-examples. Experimental results demonstrate an overall accuracy of 94% for the proposed automated triage framework, which corresponds to a 40% improvement over conventional scripting methods.

I. INTRODUCTION

Recent market studies and technical road-maps confirm the criticality and galloping resource demands of verification in modern VLSI design flows. Today, up to 46% of the design cycle is spent in efforts to verify system functionality [1]. With *debugging* constituting a major component that consumes up to 32% of the verification process [1], the semiconductor industry is at a constant lookout for Computer-Aided Design (CAD) solutions to automate the aforementioned time-demanding tasks.

Debugging commences once a discrepancy between specification and simulation is discovered, expressed in the form of a counter-example trace. Cutting-edge automated debuggers instrument formal methodologies to ameliorate the debugging process [2], [3]. This is accomplished as tools utilize counter-examples to generate a set of design locations that can explain the erroneous behavior. These locations provide vital suggestions to the engineer as to where the actual error lies in the design. However, regression verification flows complicate and prolong the debugging task, since they can potentially generate hundreds of counter-examples to be fixed. At the end of regression tests, knowledge of the relation between counter-examples and their culprit is limited. Normally, this causes confusion in the design and/or verification engineering team, since the design's erroneous behavior can be comprehensive by some of the engineers but totally opaque to others. Consequently, the problem is constantly assigned and re-assigned to various engineers until the most suitable one is found. Additionally, if the resulting set of counter-examples is not appropriately categorized into smaller related groups, then multiple engineers might be investing effort to resolve the same design error; a waste of precious resources.

Along these lines, *trriage* is the high-level debugging task following regression verification that has a twofold purpose. First, it tries to determine the relation between generated counter-examples with respect to their root cause failures. The main aim of this step is to group together those counter-examples that are closely related. Second, it aims to identify the most

suitable engineer to perform detailed debugging for each one of the formed groups. The benefit of triage lies into the fact that only those engineers familiar with the erroneous behavior will pursue detailed debugging for a given group. Moreover, any fix for a counter-example can potentially eliminate most counter-examples belonging to the same group, since they are probably caused by the same design error.

Current studies indicate that triage can potentially occupy up to 30% of the debugging effort [1]. Despite these projections, triage in modern flows is predominantly performed in an *ad hoc*, manual and time-consuming manner. In the majority of cases, triage is based on scripts that parse error messages and/or revision control information (i.e. ticketing systems) to group the observed failures. Alternatively, a single engineer is assigned to monitor and analyze error traces on a daily basis to determine the best suited engineer for further debugging. The scripting approach suffers from frequent inaccuracy in counter-example classification, whereas the manual nature of binding an engineer to the triage task incurs significant cost in terms of time and relies on the engineer's intuition and inherent understanding of the design's behavior.

In this work we present a novel automated counter-example triage framework. More precisely, the contributions are as follows. First, we devise a ranking system for possible design error locations to quantify their probability of being an actual error. This is achieved by performing a probabilistic analysis to show that errors *i*) are usually excited a small number of cycles before the observed mismatch, and *ii*) are usually covered by simulation only a small number of times before being excited. Second, we introduce the concept of *counter-example proximity*, a novel speculative metric that expresses similarity or dissimilarity between counter-examples based on the likelihood of originating from the same error source or from distinct ones. The suggested metric is constructed by exploiting, simulation coverage, satisfiability, and the proposed ranking system to determine counter-example correlation. Triage is then formulated as a pattern recognition problem and solved using hierarchical clustering. Our approach allows us to employ machine learning algorithms to build an automated debugging triage framework. The developed framework is tested on four different designs with multiple injected errors demonstrating significant gains over existing triage methodologies.

The remainder of this paper is organized as follows. Section II reviews background and presents basic concepts and notation on SAT-based design debugging along with notation on simulation coverage. Section III defines the problem of triage in debugging, and Section IV introduces the proposed failure triage framework along with suggested metrics and heuristics. Finally, Section V provides experimental results and Section VI concludes the paper.

II. PRELIMINARIES

A. Notation for SAT-based Design Debugging

This sub-section describes SAT-based design debugging and introduces relevant notation, which is used throughout the paper. Assume an erroneous design \mathcal{D} that fails verification because of a single or multiple errors in the RTL. When a mismatch between the expected values and the simulated ones in \mathcal{D} is identified at Primary Outputs (POs) we say that a failure occurs. Let also $\mathcal{C} = \{c_1, c_2, \dots, c_{|\mathcal{C}|}\}$ denote a set of counter-examples that expose $|\mathcal{C}|$ failures and $len(c_i)$ denote the length of counter-example c_i in number of cycles. The output of an automated debugger for each counter-example c_i is a set of circuit elements (RTL blocks or signals) that can be responsible for c_i . This set is referred to as a *solution* [3] for c_i denoted as $\mathcal{S}_i = \{s_{i_1}, s_{i_2}, \dots, s_{i_{|\mathcal{S}_i|}}\}$ since each circuit element $s_{i_j} \in \mathcal{S}_i$ can be modified to rectify the erroneous behavior exhibited in c_i . Any s_{i_j} is referred to as a *suspect* for counter-example c_i . The cycle where an error can be excited at s_{i_j} is also available at the output of the debugger, and is denoted as t_{i_j} . In this work, the notion of *excitation cycle*, t_{i_j} , refers only to the excitation that eventually propagates the error to an observable output. Remark that it is possible for two distinct counter-examples c_i and c_j to share one or more suspects in their solutions. The set of *mutual suspects* between c_i and c_j is denoted as:

$$(M_{ij} \equiv M_{ji}) = \{\{s_{i_k}, s_{j_w}\} : s_{i_k} = s_{j_w}\} \quad (1)$$

For uniformity, M_{ii} is also defined under Eq. 1, where all tuples contain each suspect in \mathcal{S}_i twice. Moreover, SAT-based mechanics allow debuggers to return error propagation paths in the circuit that show how a value from a suspect s_{i_j} propagates to reach the PO where a mismatch was observed [4]. Error propagation paths start from the cycle where a suspect is excited to propagate the erroneous value. All cycles before the excitation cycle are referred to as the *prefix* of the suspect, while all cycles following until the observation of a mismatch are called the *suffix* of that suspect. For example, if s_{i_j} is excited at cycle k , then $t_{i_j} = k$, $prefix(s_{i_j}) = [1 \dots k]$, and $suffix(s_{i_j}) = [k + 1 \dots len(c_i)]$.

An example of debugging a counter-example is depicted in Fig. 1 by using the Iterative Logic Array representation of the sequential design [3]. An error is excited in cycle $m - 2$ and propagates to cause a failure at the output in cycle m . The generated counter-example c_i of length $len(c_i) = m$ is then passed to an automated debugger. The result is a solution $\mathcal{S}_i = \{s_{i_1}, s_{i_2}, s_{i_3}\}$ of circuit elements that can potentially explain the wrong output. Suspects s_{i_1} , s_{i_2} and s_{i_3} , excited in cycles k , $m - 2$ and $m - 1$ respectively, along with their propagation paths are illustrated in Fig. 1. As seen in the same figure, the suffix and prefix parts of the counter-example differ among the suspect locations. Also notice that the exhaustive nature of the underlying SAT engine will definitely return the actual error location as a suspect in the solution set [3].

B. Notation for Simulation Coverage

RTL simulation can provide us with the number of times each line, block or branch in the RTL description of \mathcal{D} is executed. We refer to this number as the *frequency* of the respective circuit element. Since all suspects s_{i_j} in a counter-example solution \mathcal{S}_i correspond to circuit elements, a mapping between s_{i_j} and its corresponding frequency is always feasible. We denote this frequency as $freq(s_{i_j})$. Note that for two mutual suspects

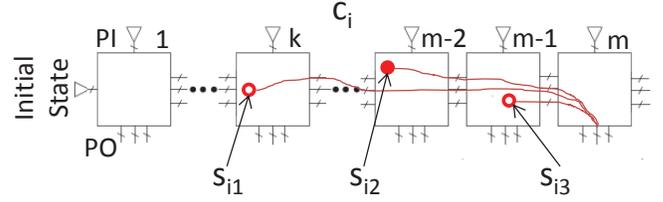


Fig. 1. Counter-example and debugging suspects

$s_{i_k} = s_{j_l}$ for counter-examples c_i and c_j it is possible that $freq(s_{i_k}) \neq freq(s_{j_l})$ when $len(c_i) \neq len(c_j)$. In other words, the same circuit element can be executed a different number of times between two distinct counter-examples of different length, even if it is a suspect location for both.

III. TRIAGE IN DEBUGGING

A. Problem Definition

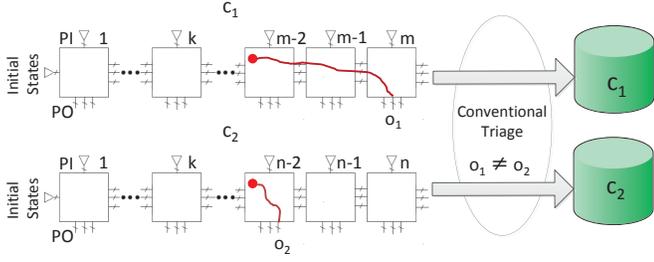
As it becomes apparent from Section II, the task of debugging a single counterexample is unequivocal; the automated debugger will return a solution set that can justify the erroneous behavior, and from that set, all suspects will be examined by the engineer to track down the actual error. Moreover, a quick overview of the suspect locations is usually adequate to identify the rightful owner that should proceed with fixing the counter-example. However, the existence of multiple counter-examples at the end of regression simulation needs to be managed by determining their relations and forming appropriate groups as discussed in Section I. In practice though, identifying relations between counter-examples is not trivial.

Conventional approaches, such as script-based grouping of error logs or manual analysis frequently fall short when it comes to identifying counter-example relationships [5]. Fig. 2 illustrates two common cases where traditional methods tend to fail. In Fig. 2(a) an error propagates due to different stimulus through different circuit elements and is eventually causing two failures at distinct POs and possibly at different cycles. The counter-examples exposing those two failures will be wrongly grouped into two separate groups, biased by the fact that the POs -and hence the error messages- are different. The opposite scenario can also happen. Fig. 2(b) illustrates two distinct errors causing a discrepancy at the same POs, by following different propagation paths. Traditionally, the failures will be put into the same group, which is not the desired result.

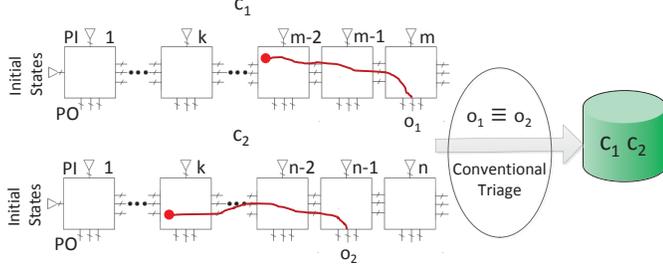
Triage addresses the aforementioned issues, by automatically grouping similar counter-examples together and passing them to the suitable engineer(s) for further root cause analysis. Using the notation and concepts presented in Section II we define failure triage as follows.

Definition 1: Given an erroneous design \mathcal{D} , and a set of counterexamples \mathcal{C} , **counter-example triage** is a complete partitioning of \mathcal{C} into a set of N subsets/groups denoted as $\mathcal{G} = \{g_1, g_2, \dots, g_N\}$ such that the following rules apply:

- **jointly exhaustive property:** There is no counter-example $c_i \in \mathcal{C}$ that does not belong to some g_k .
- **mutually exclusive property:** Each counter-example c_i belongs exactly to one g_k .
- **relation property:** Each group g_k contains failures that have a high probability of originating from the same design error.



(a) Different outputs failing because of same error



(b) Same output failing because of different errors

Fig. 2. Incorrect grouping by conventional techniques

In order for the grouping to be acceptably accurate, three critical aspects of the problem should be addressed. First, a well-defined metric needs to be devised that will quantify the relationship between two given counter-examples. Second, an estimation has to be made on the number of errors causing the whole set of counter-examples, because this determines the number of groups to be formed. Finally, an efficient technique has to be applied to form closely related groups by using the similarity metric between all possible pairs of counter-examples. The following sections describe our work to address the aforementioned issues.

IV. COUNTER-EXAMPLE TRIAGE FRAMEWORK

A. Error Behavior Analysis

From Section II.A, it becomes obvious that SAT-based debugging guarantees that an error will be returned as a suspect. However, not all suspects are real errors. As such, before constructing any triage metrics, it is crucial to identify those suspects that are likely to be real errors or relate to ones. We address the above by speculating on the way an actual error is excited and eventually propagates to the failing outputs. Suspects that follow our assumptions on how an error behaves are promoted against suspects that violate our expectations.

Generally, we expect errors to be excited in temporal proximity to the failing outputs, an intuitive argument that is central behind Bounded Model Debugging [6]. Moreover, we expect that for a design error to be excited it would take a relatively small number of times for the code it resides in to be executed (covered by simulation). In support of the argument above, we outline a probabilistic analysis.

Assuming that an error exists in the design and that simulation starts at cycle 1, let ex_i be the probability that the error is excited at cycle i . Also, let pr_i be the probability of the error propagating from cycle i to cycle $i+1$, and ob_i be the probability of observing a failure at the POs at cycle i given that the error has propagated

to that cycle. Also assume that the input vector sequences are temporally independent and stationary random sequences.

Proposition 1: The probability of observing the first failure at cycle m given the probability that the error is excited for the first time at cycle n is:

$$p_m = \prod_{i=1}^{n-1} (1 - ex_i) \times ex_n \times \prod_{i=n}^{m-1} pr_i \times \prod_{i=n}^{m-1} (1 - ob_i) \times ob_m$$

Proof: Let events:

$E_i = \{\text{an error is excited at cycle } i\}$,

$X_i = \{\text{an error propagates from cycle } i \text{ to cycle } i+1 \text{ given that it has propagated to cycle } i\}$,

$O_i = \{\text{a failure is observed in cycle } i \text{ given that an error has propagated to cycle } i\}$.

Probability p_m can be expressed in terms of the events E_i , X_i , and O_i as follows:

$$p_m = \mathcal{P}\left(\bigcap_{i=1}^{n-1} \overline{E}_i \cap E_n \cap \left(\bigcap_{i=n}^{m-1} X_i \cap \bigcap_{i=n}^{m-1} O_i \cap O_m \mid E_n\right)\right).$$

But events $\bigcap_{i=1}^{n-1} \overline{E}_i$ are conditionally independent to $\bigcap_{i=n}^{m-1} X_i$,

$\bigcap_{i=n}^{m-1} O_i \cap O_m$. Thus,

$$p_m = \mathcal{P}\left(\bigcap_{i=1}^{n-1} \overline{E}_i \cap E_n\right) \times \mathcal{P}\left(\bigcap_{i=n}^{m-1} X_i \cap \bigcap_{i=n}^{m-1} O_i \cap O_m \mid E_n\right).$$

By Bayes' law and the chain rule we have

$$\mathcal{P}(A \cap B \mid C) = \mathcal{P}(A \mid C) \times \mathcal{P}(B \mid A \cap C).$$

Hence, $p_m = \mathcal{P}\left(\bigcap_{i=1}^{n-1} \overline{E}_i \cap E_n\right) \times \mathcal{P}\left(\bigcap_{i=n}^{m-1} X_i \mid E_n\right) \times$

$\mathcal{P}\left(\bigcap_{i=n}^{m-1} \overline{O}_i \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right) \times \mathcal{P}\left(O_m \mid \bigcap_{i=n}^{m-1} O_i \cap \bigcap_{i=n}^{m-1} X_i \cap E_n\right)$.

But events O_m and $\bigcap_{i=n}^{m-1} \overline{O}_i$ are conditionally independent given E_n , thus p_m can be re-written as

$$p_m = \mathcal{P}\left(\bigcap_{i=1}^{n-1} \overline{E}_i \cap E_n\right) \times \mathcal{P}\left(\bigcap_{i=n}^{m-1} X_i \mid E_n\right) \times$$

$$\mathcal{P}\left(\bigcap_{i=n}^{m-1} \overline{O}_i \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right) \times \mathcal{P}\left(O_m \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right).$$

By assumption, inputs of consecutive cycles are temporally independent. As a result, X_i is independent of X_j , and E_i is independent of E_j for all $i \neq j$, meaning that

$$\mathcal{P}(X_i \cap X_j \mid E_n) = \mathcal{P}(X_i \mid E_n) \times \mathcal{P}(X_j \mid E_n), \text{ and}$$

$$\mathcal{P}(E_i \cap E_j) = \mathcal{P}(E_i) \times \mathcal{P}(E_j). \text{ Consequently,}$$

$$\mathcal{P}\left(\bigcap_{i=n}^{m-1} X_i \mid E_n\right) = \prod_{i=n}^{m-1} \mathcal{P}(X_i \mid E_n), \text{ and}$$

$$\mathcal{P}\left(\bigcap_{i=1}^{n-1} \overline{E}_i \cap E_n\right) = \prod_{i=1}^{n-1} \mathcal{P}(\overline{E}_i) \times \mathcal{P}(E_n).$$

Similarly, conditional independence between O_i and O_j yields

$$\mathcal{P}\left(\bigcap_{i=n}^{m-1} \overline{O}_i \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right) = \prod_{i=n}^{m-1} \mathcal{P}\left(\overline{O}_i \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right).$$

Hence, p_m can be simplified to:

$$p_m = \prod_{i=1}^{n-1} \mathcal{P}(\overline{E}_i) \times \mathcal{P}(E_n) \times \prod_{i=n}^{m-1} \mathcal{P}(X_i \mid E_n) \times$$

$$\prod_{i=n}^{m-1} \mathcal{P}\left(\overline{O}_i \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right) \times \mathcal{P}\left(O_m \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right).$$

Based on the assumptions, $1 - ex_i = \mathcal{P}(\overline{E}_i)$, $ex_n = \mathcal{P}(E_n)$,

$pr_i = \mathcal{P}(X_i \mid E_n)$, $ob_i = \mathcal{P}\left(O_i \mid \bigcap_{i=n}^{m-1} X_i \cap E_n\right)$, therefore p_m

can be defined as:

$$p_m = \prod_{i=1}^{n-1} (1 - ex_i) \times ex_n \times \prod_{i=n}^{m-1} pr_i \times \prod_{i=n}^{m-1} (1 - ob_i) \times ob_m$$

■.

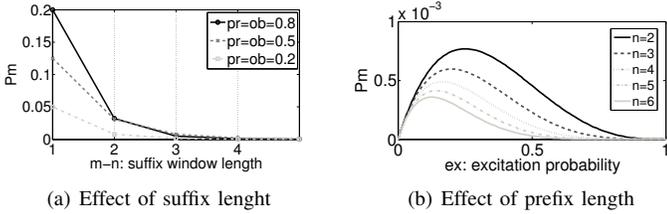


Fig. 3. Probabilistic behavior of errors

Since we simply aim to construct a generalized view of error behavior, we can assume that $pr_i = pr$, $ob_i = ob$ and $ex_i = ex$ remain the same for all cycles i . Then we get:

$$p_m = (1 - ex)^{n-1} \times ex \times pr^{m-n} \times (1 - ob)^{m-n} \times ob \quad (2)$$

Fig. 3 illustrates the results of plotting Eq. 2. To show our findings, p_m is plotted under two different settings. In the first, depicted in Fig. 3(a), the cycle where the error is first excited is kept constant such that $n = 1$, whereas cycle m where the failure is observed is selected from the set $[2, 3, 4, 5, 6]$. In other words, the prefix is set to a constant size of 1 and the suffix length varies. Additionally, the propagation, observation and excitation probabilities are set constant, such that $pr = ob = [0.2, 0.5, 0.8]$ and $ex = 0.5$. Probability p_m is plotted as a function of m .

In the second setting, depicted in Fig. 3(b), the number of cycles between the first excitation cycle and the cycle where the error is observed is kept constant so that $m - n = 2$, while n (the cycle of first excitation) takes values from the set $[2, 3, 4, 5, 6]$. Essentially, the prefix length now varies whereas the suffix has a fixed length of 2 cycles. Probabilities pr and ob are set constant to 0.8. In the second setting, p_m is plotted as a function of ex .

In Fig. 3(a), the negative exponential nature of the probability curves confirms the expectations that an error usually causes a failure only a small number of cycles after it has been excited. Hence, the error's suffix is expected to be relatively short. Fig. 3(b) leads us to an additional observation. We observe that as the prefix length increases, the highest failure observation probability, p_m , is achieved when the excitation probability becomes smaller. The above behavior confirms our intuition that even when an error is first excited close to the failure point, the longer the prefix is, the harder to excite the error it should be (its optimal excitation probability drops). Since the excitation probability is proportional to the likelihood of the error location being executed (covered by simulation), any conclusions related to excitation can be applied to its coverage as well.

It has to be noted that the description above serves not as a theoretical proof of the behavior of errors but only as an experimental intuition of the most typical cases. This is because it is based on certain assumptions. However, this probabilistic analysis leads to the following general observations about the behavior of errors in most cases. An error is expected (i) to have a relatively long prefix (short suffix), and (ii) to be covered only a small number of times during its prefix.

The above observations form the basis of the suspect ranking scheme described in the next sub-section, which guides triage metrics to more accurate outcomes, as demonstrated by our experiments.

B. Suspect ranking

For a counter-example c_i , the returned solution set S_i contains all possible suspects for the observed mismatch. However, there

are many cases where the solution set is large. To add more pain, some of the returned suspects are incidental and atypical of common error locations, such as reset signals and PI suspects. Along these lines, the utter goal of suspect ranking is to generate a ranked version of the solution that serves two purposes. First, it segregates suspects that are atypical of common errors from suspects that are actual errors or are closely related to ones. As such, counter-example relation is defined based on the latter and not by treating all suspects evenly. Second, it aids engineers to prioritize detailed debugging by first examining those suspects high in rank.

In order to generate such a ranking, we quantify the observations of Section IV.A. Recall that $freq(s_{i_j})$ is the number of times suspect s_{i_j} is covered by simulation in its prefix, $prefix(s_{i_j})$. Let $score(s_{i_j})$ be a scoring function quantifying the likelihood of s_{i_j} being an actual error, defined as follows:

$$score(s_{i_j}) = \frac{|prefix(s_{i_j})|}{len(c_i)} \times \left(1 - \frac{freq(s_{i_j}) - \gamma}{max\{freq(s_{i_k}) : s_{i_k} \in S_i\}} \right) \quad (3)$$

The higher $score(s_{i_j})$ is, the more typical of an actual error suspect s_{i_j} is, always based on our probabilistic analysis. The first factor $\frac{|prefix(s_{i_j})|}{len(c_i)}$ in Eq. 3 is in the range of $[0 \dots 1]$ and quantifies the expectation that a real error is excited in temporal proximity to the observed mismatch. The longer $prefix(s_{i_j})$ is, the higher $score(s_{i_j})$ becomes, as desired. The second factor increases as coverage of s_{i_j} prior to the excitation cycle decreases, respectively resulting in an increase to the score function, again as desired. Similar to the first factor, the second also falls within the range of $[0 \dots 1]$ for homogeneity. The denominator is set to the maximum coverage observed for all suspects in the corresponding solution, as a measure of comparison. A relatively small offset γ is subtracted from the numerator to avoid zeroing out the contribution of suspects that have maximum coverage for the counter-example.

Based on the scoring function above we can construct a ranking for all suspects. Let $rank$ be a *relation*, such that for two distinct suspects s_{i_j} and s_{i_k} , if $rank(s_{i_j}) < rank(s_{i_k})$, then s_{i_j} is more likely to be the actual error compared to s_{i_k} , respectively $score(s_{i_j}) > score(s_{i_k})$. Given that $score(s_{i_j})$ has been computed for all $s_{i_j} \in S_i$, $rank(s_{i_j})$ is defined as:

$$rank: \{score(s_{i_j}) : s_{i_j} \in S_i\} \rightarrow \{1, 2, \dots, |S_i|\} \\ rank(s_{i_j}) = \{r : |\{s_{i_k} \in S_i : score(s_{i_k}) \geq score(s_{i_j})\}| = r - 1\} \quad (4)$$

Based on the above equations, real errors and suspects related to them are more likely to be placed high in rank, exactly as desired.

C. Counter-example proximity

The cornerstone of triage is a well-defined metric to express relation between any two given counter-examples. In order to develop such a metric we exploit information from the suspect ranking scheme along with the number of suspects that two counter-examples share in common.

As defined in Section II.A, the set of mutual suspects between two counter-examples c_i and c_j is denoted as M_{i_j} . Intuitively, when M_{i_j} is large relative to the number of total suspects in both solutions, then c_i and c_j are considered to be strongly related, thus possibly originating from the same error source. However, if mutual suspects are low in ranking in both or at least one of

the solutions, then this correlation becomes weaker. For example, if a mutual suspect is high ranked in S_i but low ranked in S_j , then it is more likely to be a real error for counter-example c_i and not for c_j , even if it can fix both; counter-example c_j is expected to be caused by a suspect higher in rank. We combine the above expectations into a speculative metric called counter-example proximity between any pair c_i and c_j , which is denoted as $prox(c_i, c_j)$ and defined as:

$$prox(c_i, c_j) = 1 - \frac{|M_{ij}|}{|S_i| + |S_j| - |M_{ij}|} \times \prod_{\{s_{i_k}, s_{j_w}\} \in M_{ij}} \left(1 - \frac{|rank(s_{i_k}) - rank(s_{j_w})|}{\max\{|S_i|, |S_j|\}} \right) \quad (5)$$

According to Eq. 5, when c_i and c_j are strongly related then $prox(c_i, c_j)$ tends to 0, whereas a weak correlation sets $prox(c_i, c_j)$ closer to 1. Remark, that the number of mutual suspects over total suspects is encoded in the factor $\frac{|M_{ij}|}{|S_i| + |S_j| - |M_{ij}|}$. As desired, a large mutual suspect set M_{ij} will force $prox(c_i, c_j)$ closer to 0. In the case where all suspects in solutions S_i and S_j are mutual then $|M_{ij}| = |S_i| = |S_j|$, thus $\frac{|M_{ij}|}{|S_i| + |S_j| - |M_{ij}|} = 1$, and the first factor maximizes its contribution. In this context, the second factor quantifies the contribution of mutual suspects based on their ranking. Ideally, counter-examples caused by the same error will exhibit similar behavior. Therefore, their mutual suspects are expected to have similar ranks in their respective solution sets. Based on Eq. 5, proximity decreases as the difference $|rank(s_{i_k}) - rank(s_{j_w})|$ in the ranking of mutual suspects increases, which models the above expectation. Remark that, $prox(c_i, c_i) = 0$ as desired, since all suspects are mutual ($\frac{|M_{ij}|}{|S_i| + |S_j| - |M_{ij}|} = 1$) and have the same rank ($|rank(s_{i_k}) - rank(s_{j_w})| = 0$ always). On the other hand, if c_i and c_j share no mutual suspects then they are definitely unrelated and caused by different errors, which is successfully captured by Eq. 5, since in that case $|M_{ij}| = 0$ and thus $prox(c_i, c_j) = 1$.

D. Error Count Estimation

For a grouping of the generated counter-examples to be meaningful, it is necessary to define the number of groups expected to be formed. Ideally, this number should equal the number of design errors responsible for the whole set of counter-examples \mathcal{C} . However, in the vast majority of regression scenarios the number of co-existing errors is not known *a priori*. Therefore an initial *guess* on the number of groups has to be made that will reflect an acceptable grouping scheme.

For that purpose, we construct a heuristic called *error count estimation* that leverages information from the suspect ranking scheme. Each mutual suspect set M_{ij} is reduced to set M_{ij}^R which contains *only* those suspects that have *at most* a rank of $R \leq \min\{|S_i|, |S_j|\}$ in suspect sets S_i or S_j . Formally:

$$M_{ij}^R = M_{ij} \setminus \{ \{s_{i_k}, s_{j_w}\} \in M_{ij} : (rank(s_{i_k}) > R) \vee (rank(s_{j_w}) > R) \} \quad (6)$$

Intuitively, high-ranked suspects (small R) are closely related to actual errors, thus a large number of such mutual suspects indicates that counter-examples are caused by a small number of errors, and vice versa. After computing all possible sets M_{ij} and the reduced sets M_{ij}^R , for each suspect we generate a set

containing all its manifestations that appear to a reduced mutual set, denoted as $count_{s_{i_k}}$:

$$count_{s_{i_k}} = \{s_{j_w} : \{s_{i_k}, s_{j_w}\} \in M_{ij}^R\} \quad (7)$$

For example, assume some s_{1_3} has rank R or less in 3 different counter-examples c_1, c_2 and c_4 , where it appears as s_{2_2} and s_{4_3} in c_2 and c_4 . Recall that M_{ii} contains $\{s_{1_3}, s_{1_3}\}$ by definition. Then $count_{s_{1_3}} = \{s_{1_3}, s_{2_2}, s_{4_3}\}$ and $|count_{s_{1_3}}| = 3$. Remark that $count_{s_{1_3}} = count_{s_{2_2}} = count_{s_{4_3}}$. As such, let \widehat{count} be a set that contains only one copy of all computed $count_{s_{i_j}}$ sets. The average number of times such high-ranked suspects participate in a solution set estimates how many counter-examples we expect those suspects to be responsible for, on average:

$$count_{avg} = \frac{\sum_{count_{s_{i_j}} \in \widehat{count}} |count_{s_{i_j}}|}{|\widehat{count}|} \quad (8)$$

Then, our error count estimation, denoted as e , is given by:

$$e = \left\lceil \frac{|\mathcal{C}|}{count_{avg}} \right\rceil \quad (9)$$

Eq. 9 essentially says that the expected number of co-existing errors responsible for all counter-examples is calculated by dividing the number of counter-examples $|\mathcal{C}|$ by the average number of counter-examples we expect each high-ranked suspect to be responsible for. Observe that, if no mutual suspects of high rank (R) exist, then $|count_{s_{i_j}}| = |\{s_{i_j}\}| = 1$ for all high-ranked suspects, and $count_{avg} = 1$ according to Eq. 8. Then the error count estimation will be $e = |\mathcal{C}|$, acceptably predicting that each counter-example was most likely caused by a unique error, and thus the number of error equals the number of counter-examples. On the other hand, the existence of high-ranked suspects among various solutions incurs a decrease in e , since $count_{avg}$ increases. Eq. 9 offers a loose approximation on the number of co-existing errors, but is sufficient to guide the formation of groups as demonstrated by experimental results.

E. Overall Flow

The information embedded in the metrics described above is applied for the last step of the proposed triage framework, which is the formation of groups of similar counter-examples. For that purpose, we formulate triage as a clustering problem and employ a hierarchical clustering algorithm [7] to solve it.

Hierarchical clustering aims to group together elements based on their relationship, which is quantified by a metric called *distance*. A distance metric usually takes real values, and is assigned per pair of elements. If the distance between a pair is small then the elements are considered strongly related and vice versa. In our framework, the elements to be grouped are essentially counter-examples. Based on the definition of counter-example proximity, its use as a distance metric is appropriate, since it follows the desired properties described above.

In the context of our work, hierarchical clustering takes as input the set of all counter-examples \mathcal{C} , and the proximity between all pairs of counter-examples in the form of a $|\mathcal{C}| \times |\mathcal{C}|$ matrix. The output is not a single grouping. Rather, this algorithm generates all possible groupings of \mathcal{C} . However, the error count estimation presented in this section, suggests the most reasonable one based on our pre-processing and assumptions made. Groups

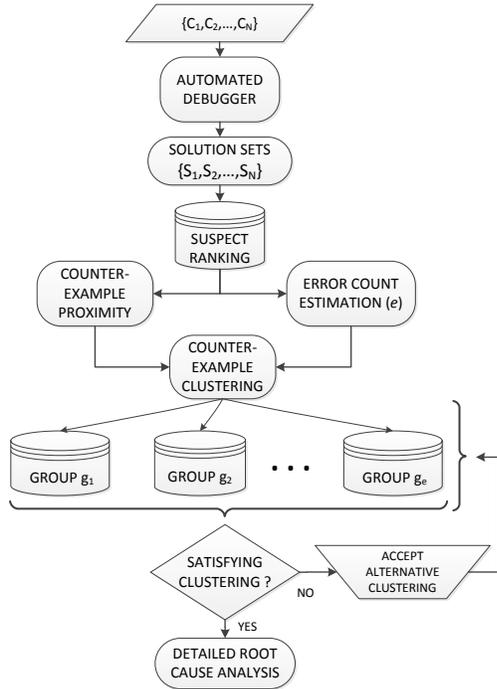


Fig. 4. Proposed triage framework

of failures are formed in a bottom-up fashion (agglomerative) by merging clusters that are likely to contain counter-examples that are related. The decision to merge two clusters is determined by a linkage criterion. In this work, we use Ward’s Method [7], where at each step we merge the pair of clusters that leads to minimum increase in total within-cluster variance after merging. Ward’s method tends to create compact clusters, which proved to perform well, as shown by experiments in the next Section.

Fig. 4 illustrates the overall flow that contains the steps described above in this section. The input to the flow is a set of counter-examples generated by regression verification. The debugger is evoked and provides a solution set for each counter-example. Based on Eq. 3, a ranked version of the suspects is constructed. The ranking scheme is subsequently utilized for the computation of counter-example proximity and the error count estimation based on Eq. 5 and Eq. 9 respectively. Those metrics are then passed to the clustering algorithm that forms all possible clusterings of related counter-examples. The main output of the triage engine is the unique clustering that comprises of e related groups, suggested by the error count estimation. The grouping is then examined by engineers, along with the suspect ranking scheme which is already computed. Remark that the triage process is initially executed with the error count estimation, but it depends on the engineer to accept the formation of e groups or examine an alternative number of groups already computed by the engine.

V. EXPERIMENTAL RESULTS

This section presents preliminary experimental results for the proposed triage framework. All experiments were conducted on a single core of an Intel Core i5 3.1 GHz workstation with 8GB of RAM. Four *OpenCores* [8] designs were used for the evaluation. The underlying automated debugging tool used for extracting the suspect locations was implemented based on [3]. A platform

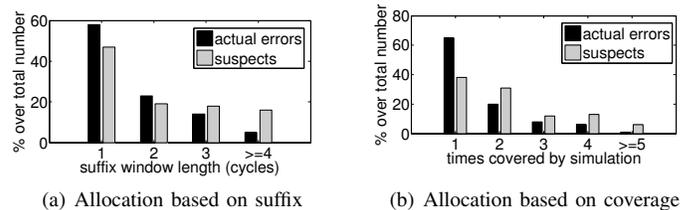


Fig. 5. Features of real errors and suspects across all testcases

coded in Python was developed to parse the returned results of the debugger, calculate the relevant metrics and perform hierarchical clustering on the resulting counter-examples. For each design, a set of different errors was injected each time by modifying the RTL description. In total, sixteen regression simulations were run, generating a different number of counter-examples each time, caused by a different set of errors.

A first set of experiments was conducted to confirm the claims made based on our probabilistic analysis in Section IV.A. After regression simulation, 285 counter-examples were collected for all designs, and since the actual error was known between the returned suspects, we observed their first excitation cycle along with the frequency of the corresponding RTL component. Results are illustrated in Fig. 5, where we see that both real errors and suspects generally follow our expectations that suffix length is generally short and that such locations have a small frequency in their prefix. However, real errors tend to follow the above pattern more accurately compared to the rest of the suspects; a feature that enables real errors to be generally high in the suspect ranking scheme and comes in compliance with our expectations.

A second set of experimental results is depicted in Fig. 6, where we explore the effect of the ranking scheme on the framework’s accuracy across all testcases. The ratio $(1 - \frac{\text{misclassified } C_i\text{'s}}{|C|})$ determines accuracy. A counter-example belonging to the wrong group is considered misclassified. In Fig. 6, $R = 0$ denotes the absence of ranking scheme ($M_{ij}^R = M_{ij}$ always). More precisely, Fig. 6(a) demonstrates how bigger the estimate e is, which is automatically computed by the engine, when compared to the actual number of errors. Apparently, computations devoid of any suspect ranking lead to the inclusion of 2.8 extra clusters on average. This reflects to a 77% average accuracy for the triage engine shown in Fig. 6(b). Remark, that selecting only the top ranked suspect ($R = 1$) discards useful knowledge by excluding the rest of the suspects and results into 3.1 extra clusters on average, decreasing overall accuracy to 74%. Also remark that, when R is set too high, low rank suspects are included in the computations and introduce noise to the error count estimation. This also incurs a decrease in overall accuracy shown in Fig. 6(b). However, any reasonable selection for R , between 2 – 10, results in better accuracy overall, with the best outcome of 94% achieved when $R = 4$. Note that in the latter case, e is off only by 0.7 on average. Generally, even for extreme values of R (1 or 10) the triage engine always outperforms conventional scripting-based triage, which achieves a 67% average accuracy shown by the straight line in Fig. 6(b).

Table I demonstrates detailed results for all sixteen testcases and four designs with $R = 4$ so that the top 4 in rank suspects are selected for the error count estimation. Our selection is indicative as we choose this value of R because it reflects a good behavior for the algorithm. Again, though, any reasonable $R \in [2 \dots 10]$ generates similar results as shown in Fig. 6. The first and second columns refer to the design name and its size respectively.

TABLE I
PROPOSED TRIAGE ENGINE PERFORMANCE

circuit	# gates	# errors	C	e	accuracy			# suspects (avg)	error rank (high - low)	time (sec)
					trriage(e)	trriage(# errors)	script			
fpu	83303	4	15	4	100%	100%	80%	12.5	1-6	12.8
		5	20	6	83%	100%	65%	14.1	2-5	13.8
		6	24	6	100%	100%	68%	12.7	1-4	16.7
		7	31	7	95%	95%	65%	14.3	1-7	19.8
vga	72292	3	14	4	88%	100%	71%	14.0	2-6	14.3
		4	15	6	78%	89%	67%	13.9	2-9	15.5
		5	22	5	100%	100%	55%	15.1	1-4	17.9
		6	29	7	89%	95%	69%	12.9	1-5	19.3
spi	1724	2	8	2	100%	100%	75%	10.8	1-3	12.4
		3	13	3	100%	100%	62%	11.4	1-4	12.7
		4	16	5	90%	100%	63%	12.2	1-5	13.4
		5	16	6	94%	100%	69%	10.3	1-5	15.7
mem_ctrl	46767	4	9	4	100%	100%	56%	15.8	1-6	12.2
		5	15	5	100%	100%	67%	14.6	1-4	12.7
		6	18	8	89%	94%	72%	15.0	2-4	13.8
		7	20	9	90%	95%	70%	15.2	1-5	15.7
				AVG:	94%	98%	67%			14.9

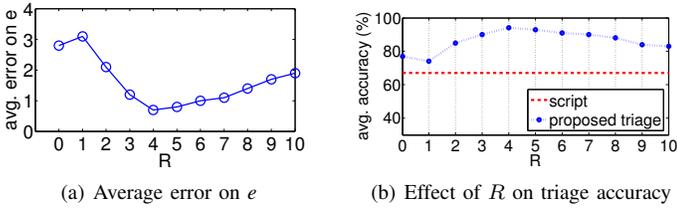


Fig. 6. Effect of R on triage across all testcases

Columns 3 and 4 contain the actual number of errors that were injected into the design and the number of total counter-examples generated. Column 5 shows the error count estimation that the proposed method generates for each testcase. Columns 6 to 8 include a comparison in accuracy between the proposed triage flow and a typical binning strategy based on a script that exploits error message information. Specifically, columns 6 and 7 refer to the accuracy of the triage flow when performed with our error count estimation or with the actual number of errors respectively, assuming prior knowledge of that number. The ninth column presents the average number of suspects across all counter-examples in each regression session. The tenth column presents the lowest and highest rank assigned to the actual error in the ranking list. The last column indicates the total time consumed by SAT-based debugging, the calculation of the two metrics, and the clustering process.

The engine’s average accuracy reaches 94% when the algorithm is executed with our initial guess (column 6) and reaches 98% for those groupings where the number of clusters equals the number of design errors. Generally, a perfect initial guess that reflects to the actual number of errors was observed in seven out of sixteen testcases, achieving a 99% accuracy on average. On the other hand, in cases where the error count estimation is off by one or two clusters, accuracy drops to 88%. A conventional approach similar to the one in Section III consisting of scripts to perform the grouping achieves an overall accuracy of 67%. As such, the proposed method improves accuracy up to 40% when the initial guess is utilized; a solid improvement that indicates the potential of the proposed framework. Moreover, the actual error is assigned a high rank in the suspect ranking list, as shown in column 10. Finally, computation of the two metrics and clustering consume an average of 14.9 seconds in total, which is acceptable for the purposes of triage.

It should be noted that one limitation of this work is that for errors to be identified as significant suspects they need to be relatively easy to excite and easy to propagate to observation points. Human introduced errors that manifest as stuck-at faults, bit flips, or wrong gates usually misguide the triage engine. However, in reality, such cases are less frequent.

VI. CONCLUSION AND FUTURE WORK

In this work, a novel automated debugging triage framework is proposed. The algorithm extracts information from simulation and debugging results to define relationship between various counter-examples. Strongly related counter-examples are then grouped together to guide detailed debugging. In order to quantify counter-example relation we introduce the concept of counter-example proximity and propose a suspect ranking scheme for its computation. Furthermore, we devise a speculative metric to estimate the number of co-existing errors. The efficacy of the triage engine is demonstrated by experiments within typical regression verification flows, indicating a significant increase in grouping accuracy compared to traditional triage techniques. One interesting extension to this work would be to utilize information from passing verification iterations where no counter-examples are generated, in order to enhance knowledge about suspect locations.

REFERENCES

- [1] H.Foster, “From volume to velocity: The transforming landscape in function verification,” in *Design Verification Conference*, 2011.
- [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] B. Keng and A. Veneris, “Path directed abstraction and refinement in sat-based design debugging,” in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC ’12, 2012.
- [5] S.Safarpour, B.Keng, Y.S.Yang, and E.Qin, “Failure triage: The neglected debugging problem,” in *Design and Verification Conference (DVCON)*, 2012.
- [6] S. Safarpour, A. Veneris, and F. Najm, “Managing verification error traces with bounded model debugging,” in *ASP Design Automation Conf.*, 2010.
- [7] G. J. Szekeley and M. L. Rizzo, “Hierarchical clustering via joint between-within distances: Extending ward’s minimum variance method,” *Journal of Classification*, vol. 22, no. 2, pp. 151–183, 2005.
- [8] OpenCores.org, “<http://www.opencores.org>,” 2007.