A Failure Triage Engine Based On Error Trace Signature Extraction

Zissis Poulos[1], Yu-Shen Yang[2] Andreas Veneris[1]

[1]Dept. of ECE, University of Toronto, Toronto, Canada. {zpoulos, veneris}@eecg.toronto.edu
[2]Vennsa Technologies Inc., Toronto, Canada, terry@vennsa.com

***Abstract*—The ever growing demand for functionally robust and error-free industrial electronics necessitates the development of techniques that will prohibit the propagation of functional errors to the final tape-out stage. This paramount requirement in the semiconductor world is imposed by the equivocal observation that functional errors slipping to silicon production introduce immense amounts of cost and jeopardize chip release dates. Functional verification and debugging are burdened with the tedious task of guaranteeing logic functionality early in the design cycle.**

In this paper, we present an automated method for the very first stage of functional debugging, called *failure triage*. Failure triage is the task of analyzing large sets of failures, grouping together those that are likely to be caused by the same design error, and then allocating those groups to the appropriate engineers for fixing. The introduced framework instruments techniques from the machine learning domain combined with the root cause analysis power of modern SAT-based debugging tools, in order to exploit information from error traces and bin the corresponding failures using clustering algorithms. Preliminary experimental results indicate an average accuracy of 93% for the proposed failure triage engine, which corresponds to a 43% improvement over conventional automated methods.

***Index Terms*—Failure Triage, Design Debugging, Regression Tests, Clustering**

I. Introduction

The galloping pace at which integrated circuit designs grow in size and complexity has inevitably led to a struggle in the semiconductor industry to introduce full automation in each level of the design process. However, Register Transfer Level (RTL) design abstraction still suffers from human introduced errors that can manifest themselves in a manifold of ways during various design stages. As such, in typical VLSI CAD flows, engineers spend a considerable amount of time in simulation, verification and rigorous testing before tape out. Since verification constitutes up to 70% of the time needed for design cycles, with debugging dominating the verification effort [1], industry is at a constant lookout for advanced and sophisticated automated debugging strategies.

State of the art debugging tools employ powerful formal engines to ease the debugging effort [2], [3], [4]. When a failure is discovered during verification, automated root cause analysis specifies a set of possible error locations in the design. This valuable information aids the engineer to discover the actual cause of the failure. However, in modern IC designs that undergo development, nightly regression tests usually result in a large set of failures. If those failures are not thoroughly assigned to the rightful engineers for fixing, then the engineer's lack of familiarity with the erroneous behaviour might nullify the benefits of the automated debugger.

In the context of regression verification flows, *failure triage* is defined as the proccess of determining failures that are likely to share the same root cause, grouping them together, and passing those groups to the appropriate engineer(s) for fixing. Today, there are two widely adopted approaches for failure triage. The first is to assign the task to a single engineer, who's responsibility is to monitor and analyze failures on a daily basis, and determine the best suited engineer for further root cause analysis. Indisputably, the efficacy of this strategy highly relies on the engineer's intuitive skills and inherent understanding of the system's behaviour. The second approach is fully automated and refers to the use of scripts to bin failures based on their corresponding error messages. Normally, the main downside of the first approach is its manual nature and cost in terms of time, whereas the latter automated one suffers from frequent inaccuracy in failure classification.

In this work we provide, for the first time, a formal definition of failure triage in a debugging context, and a failure triage framework for the first phase of design debugging. More specifically, the contributions of this work are as follows. First, we introduce the concept of *failure proximity*, a speculative metric for determining the similarity or dissimilarity between failures, regarding the likelihood of sharing the same root cause. Our method is based on exploiting error signatures from the resulting error trace after simulation. Those signatures are extracted from error excitation and propagation paths that are determined by modern design debugging tools. Second, we describe a heuristic to guess the actual number of co-existing errors in the design when no prior knowledge is assumed. In other words, our heuristic tries to predict the number of non-overlapping groups that have to be formed during triage. Last, we employ an hierarchical clustering algorithm to combine the extracted information and classify the various failures appropriately. The developed failure triage framework is tested on three different designs, to demonstrate its efficiency and applicability.

The remainder of this paper is organized as follows. Section II reviews background and presents basic concepts on satisfiability based design debugging. Section III defines the problem of failure triage, and introduces the proposed failure triage framework along with our new metrics and heuristics. Finally, Section IV provides experimental results and Section V concludes the paper.

II. Preliminaries

A. Design Debugging

Design debugging commences once a failure occurs during RTL verification. This failure is usually caught by testbench simulation using checkers at the output of the circuit and various formal properties written in an assertion language (i.e. SystemVerilog Assertions, Property Specification Language). The goal of functional debugging is to find a set of error locations, or *suspects*, which could potentially be responsible for the observed failure during verification [3]. Satisfiability-based (or SAT-based) debuggers formulate the problem as a SAT instance for the resulting error trace [4]. Such debuggers return sets of error locations in the design where a change can fix the erroneous behaviour for the given error trace. These locations can be internal signals or RTL modules. Moreover, modern mechanics allow the automated debugger to additionally return *error propagation paths*. Error propagation paths show exactly how an excited error propagates through elements of the circuit to reach the point where the checker was triggered or the assertion failed. Note, that each suspect location belongs to one or more error propagation paths, since there are potentially many ways that an error can cause a failure. Apparently, the set of suspects

returned by the debugger is an over-approximation of the actual error location; the exhaustive search of the underlying SAT engine will guarantee that the actual error belongs to the returned set of suspects.

The deep details of SAT-based debugging will not concern us here. Rather, we are interested in exploiting the output of the automated debugger. In what follows we provide notation that will be used throughout this paper when referring to design debugging concepts.

B. Notation

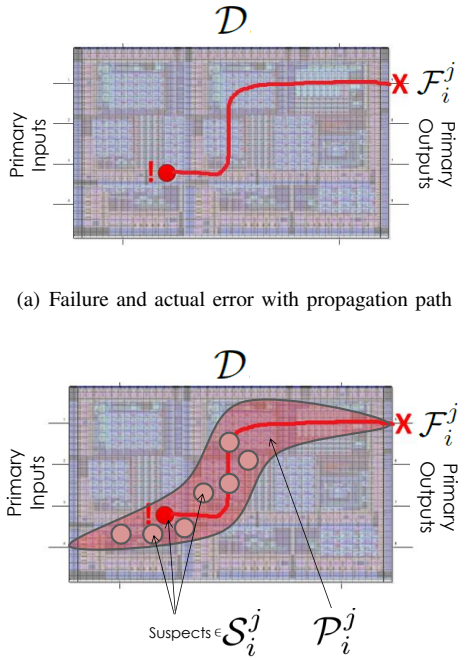
Let's assume an erroneous design, $\mathcal{D}$, that fails verification because of a single or multiple errors in the RTL. Let also $\mathcal{E}$ denote the corresponding error trace that demonstrates how the failure occurs. Also assume that the design includes a set of checkers at the Primary Outputs (POs) and various assertions on different internal signals. For the purposes of catching a failure, assertions can be viewed as more complex checkers. Therefore, for the remainder of the paper, the notion of checkers will include assertions as well. When a mismatch between the expected values and the simulated ones in $\mathcal{E}$ is identified at a checker, we say that the checker is *active* for error trace $\mathcal{E}$. Let us use set $\mathcal{C} = \{c_1, c_2, \ldots, c_{|\mathcal{C}|}\}$ to denote all $|\mathcal{C}|$ active checkers in the design under test. Note that for regression flows, usually multiple checkers will be active for $\mathcal{E}$, corresponding to many failure points during simulation. For each active checker c_i there is a set of *activation times* $\mathcal{T}_i = \{t_i^1, t_i^2, \ldots, t_i^{|\mathcal{T}_i|}\}$ indicating $|\mathcal{T}_i|$ times at which the checker was triggered (caught the failure(s)).

According to the above, $\mathcal{D}$, $\mathcal{E}$, an active c_i, and t_i^j can uniquely define a failure $\mathcal{F}_i^j$. So $\mathcal{F}_i^j = \{c_i, t_i^j\}$, where i uniquely defines the checker that fails, and j defines the time the failure occurs. In our work, the design and the error trace are unique so we ommit them for simplicity. As such, a set of failures can be defined as $\boldsymbol{F} = \{\mathcal{F}_i^j : c_i \in \mathcal{C}, t_i^j \in \mathcal{T}_i\}$.

When the above information is passed to an automated root cause analysis tool, for each failure $\mathcal{F}_i^j$, a set of possible error locations (suspects) in $\mathcal{D}$ is returned. Let's denote that set as $\mathcal{S}_i^j$. Also, a set of error propagation paths is determined, denoted as $\mathcal{P}_i^j$.

Fig 1, shows an example of a failure occurring during simulation and the result of the debugger. The thick line refers to the actual error and its propagation path in the failing design, while the shaded area corresponds to the suspects and their propagation paths. Since the suspect set is an over-approximation of the actual error location, the actual error propagation path that led to the failure will be included in the returned result.

Note that both the suspect set $\mathcal{S}_i^j$ and propagation set $\mathcal{P}_i^j$ contain information about the time (in cycles) when a possible error was excited and at what cycles the erroneous values were propagated to the point where the checker was triggered. As one can see from Fig 1, the approximation of the actual error propagation paths can significantly aid the engineer in the search for the actual error.



(a) Failure and actual error with propagation path

(b) Error propagation paths returned by the debugger

Fig. 1. Single failure and the result of root cause analysis

III. Failure Triage

A. Problem Definition

From what was described in Section II, it becomes apparent that for single failures the task of debugging is straightforward; with some insight, the failure will be assigned to the appropriate engineer, who in turn will invoke the debugger and exploit the returned information to detect the real error. In real life scenarios though, this is not usually the case. What happens when regression verification returns dozens or even hundreds of failures?

Regression simulation of large systems usually ends with multiple checkers and assertions failing. Moreover, knowledge to determine the responsible engineer(s) for each failure is limited in most cases. Regularly, this will introduce unnecessary time overhead as the problem will be passed from one engineer to another until the rightful owner is found. To add more pain, if failures sharing the same root cause are not grouped together for fixing, then multiple engineers waste time trying to resolve the same design error. The task of determining the relationship between various failures is not trivial though.

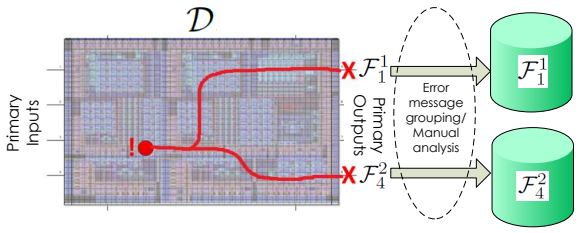
Traditional methods, such as error message grouping (using scripts) or manual analysis commonly fail to identify the relationship between pairs of failures [5]. Fig. 2 illustrates two typical cases were conventional approaches would fail. Fig. 2(a) illustrates a case where, due to different stimulus, an error propagates through different paths and is eventually captured by two different checkers c_1, and c_4, at times t_1^1 and t_4^2 respectively. Conventional techniques will wrongly group the corresponding failures separately, as there is not enough knowledge available to determine their relationship. The opposite scenario can also happen. Fig. 2(b) illustrates two distinct errors being caught by the same checker c_2, although at different times t_2^1 and t_2^2. Traditionally, the failures will be put into the same group, which is not the desired result.

Failure triage aims to alleviate the aforementioned issue, by grouping similar failures together and passing them to the suitable engineer(s) for further root cause analysis. Using the notation and concepts presented in Section II we can define failure triage as follows.

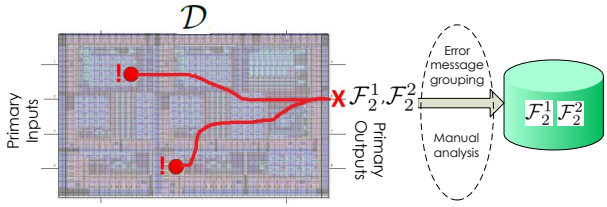
Definition 1: *Given an error trace $\mathcal{E}$ for an erroneous design $\mathcal{D}$, and a set of failures $\boldsymbol{F}$,* **failure triage** *is a complete partitioning of $\boldsymbol{F}$ into a set of N subsets/groups denoted as $\mathcal{G} = \{g_1, g_2, \ldots, g_N\}$ such that the following rules apply:*

- *There is no failure $\mathcal{F}_i^j \in \boldsymbol{F}$ that does not belong to some g_k. (jointly exhaustive)*
- *Each failure $\mathcal{F}_i^j$ belongs exactly to one g_k. (mutually exclusive)*
- *Each group g_k contains failures that have a high probability of originating from the same design error.*

Obviously, in order to produce a failure grouping that will be



(a) Different checkers failing because of same error



(b) Same checker failing because of different errors

Fig. 2. Incorrect grouping by conventional techniques

practically accurate enough, it is essential to address three major aspects of the problem:

- What defines that two failures are similar, and what the opposite.
- Approximately, how many errors co-exist in the design, so that a realistic number of groups will be produced.
- How do we form those groups by making pairwise comparisons between all failures.

The following subsections describe our work on all the above concerns.

B. Failure Proximity

In this subsection we propose a speculative metric called *failure proximity*, as a means of expressing the similarity or dissimilarity between two distinct failures. Failure proximity is generated by information that can be extracted from the error trace and the result of a simple debugging step.

For a given error trace, the way a distinct error is excited, and the way the erroneous values propagate through and affect elements of the circuit can be viewed as a *signature* for that error. Error propagation paths returned by a modern debugger can establish a correlation between the failure and the culprit. Hence, from a single error that causes two distinct failures, similar signatures will be identified, and vice versa. As a result, an effort to identify commonality between error propagation paths can potentially lead to a good criterion for distinguishing various failures with respect to their root cause.

The result of an automated debugger for the examples of Fig. 2 is illustrated in Fig. 3. Note that intersecting the error propagation paths is one simple way of identifying relationships. However, there are two observations that can refine this analysis. First, if the same checker is active for a pair of failures, then the resulting propagation paths will eventually converge towards the checker. This does not provide useful information so should be discarded. Second, if different checkers are active because of the same error, then the error propagation paths should eventually converge towards the actual error, traversing from POs to Primary Inputs (PIs). The latter, is useful information and should be promoted. Along these lines, it becomes obvious that convergence and divergence of error propagation paths along with suspect intersection is a stronger approximation of the relationship between two failures.

To identify the information described above, we perform the following steps.

For a pair of failures, say $\mathcal{F}_i^j$, $\mathcal{F}_k^l$, and their corresponding error propagation paths $\mathcal{P}_i^j$, $\mathcal{P}_k^l$:

1) Divide each propagation path into W time windows of equal length (in cycles). Let's denote as $\mathcal{P}_{i,q}^j$ ($\mathcal{P}_{k,q}^l$) those subsequences of $\mathcal{P}_i^j$ ($\mathcal{P}_k^l$) that belong to the window with index $q : 1 \leq q \leq W$.
2) For each time window q, compute the intersection of the paths, $\mathcal{P}_{i,q}^j \cap \mathcal{P}_{k,q}^l$, as well as the union $\mathcal{P}_{i,q}^j \cup \mathcal{P}_{k,q}^l$. Then generate a sequence of mutual paths over total paths per window, denoted as $\boldsymbol{M}$, where:

$$\boldsymbol{M} = \left\{ \frac{|\mathcal{P}_{i,1}^j \cap \mathcal{P}_{k,1}^l|}{|\mathcal{P}_{i,1}^j \cup \mathcal{P}_{k,1}^l|}, \frac{|\mathcal{P}_{i,2}^j \cap \mathcal{P}_{k,2}^l|}{|\mathcal{P}_{i,2}^j \cup \mathcal{P}_{k,2}^l|}, \ldots, \frac{|\mathcal{P}_{i,W}^j \cap \mathcal{P}_{k,W}^l|}{|\mathcal{P}_{i,W}^j \cup \mathcal{P}_{k,W}^l|} \right\}. \quad (1)$$

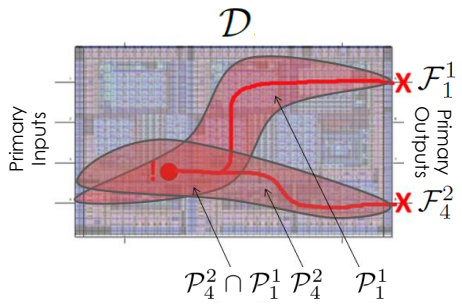
3) Detect all increasing and decreasing subsequences in $\boldsymbol{M}$, named M_{inc} and M_{dec} respectively, starting from the last window W and moving backwards in time. Then define the following function:

$$score(q) = \begin{cases} 1, & \text{if window q belongs to } M_{inc} \\ 0, & \text{if window q belongs to } M_{dec} \end{cases}$$

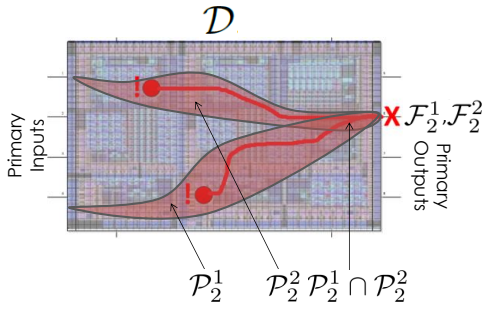
Intuitively, $\boldsymbol{M}$ contains information about the convergence or divergence of error paths from POs to PIs. In Eq.1, each ratio is within the range of $[0, 1]$, and essentially quantifies the contribution of mutual paths compared to the total number of paths that cross each window. When ratios increase from POs to PIs, this indicates convergence of error paths, and vice versa.

Then, for a pair of failures $\mathcal{F}_i^j$ and $\mathcal{F}_k^l$ our proposed failure proximity metric, denoted as $D(\mathcal{F}_i^j, \mathcal{F}_k^l)$ is given by:

$$D(\mathcal{F}_i^j, \mathcal{F}_k^l) = \sum_{q=1}^{W} \left(\frac{|\mathcal{P}_{i,q}^j \cap \mathcal{P}_{k,q}^l|}{|\mathcal{P}_{i,q}^j \cup \mathcal{P}_{k,q}^l|} \times score(q) \right) \quad (2)$$



(a) Convergence of error paths



(b) Divergence of error paths

Fig. 3. Similarity/dissimilarity of error propagation paths

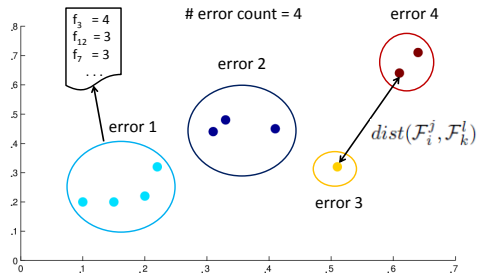


Fig. 4. Clustering for 10 failures and 4 clusters

Remark that, Eq.2 expresses failure relation by using the ratio of mutual paths over total paths per window. Moreover, the score function discards diverging paths in the calculation by zeroing out the respective intersections, while promoting converging paths. Implementation-wise, windows can belong both to the end (beginning) of a decreasing subsequence and to the beginning (end) of an increasing one. Those windows always count towards the increasing subsequence. It is easy to confirm that $D(\mathcal{F}_i^j, \mathcal{F}_k^l)$ is within the range of $[0, W]$. The closer to W the metric is, the more similar the failures are, and vice versa.

C. Error count estimation

As it will be described later, the underlying hierarchical clustering algorithm that performs the grouping of failures needs to know *a priori* the number of clusters that have to be formed. Ideally, this number should be equal to the number of the errors. However, in most cases there is no prior knowledge of how many errors are actually caught by the error trace. Therefore, an initial guess has to be made to specify a termination condition for the failure triage process.

For that purpose, we design a simple heuristic as a pre-processing step.

1) Perform pair-wise intersections $\mathcal{S}_i^j \cap \mathcal{S}_k^l$, to identify mutual suspects.
2) For each suspect s_n, record the number of failures for which the suspect is mutual. We refer to this number as *frequency* f_n of suspect s_n. If a suspect s_n is not mutual for any of the failures, the its frequency is assumed to be 1.
3) If m is the total number of distinct suspects, then from the set $\{f_1, f_2, \ldots, f_m\}$ extract the average $f_{avg} = (\sum_{i=1}^{m} f_i)/m$.

Then our error count estimation is given by:

$$e = \left\lceil \frac{|\boldsymbol{F}|}{f_{avg}} \right\rceil \tag{3}$$

Suspect frequencies provide a means of expressing how suspects are allocated among the various failures. The intuition behind this heuristic is as follows. If no mutual suspects exist then $f_{avg} = 1$ and the error count estimation will correctly predict that $e = |\boldsymbol{F}|$, because then each failure is caused by a unique error. On the other hand, the existence of mutual suspects and the number of failures those suspects are mutual for, will increase f_{avg}, and in turn will cause the error count estimation to drop, based on Eq.3. This translates to the fact that we expect some of the errors to cause multiple failures when failures share multiple common suspects. Therefore, Eq.3 offers a loose approximation of how many failures we expect to be caused by the same error *on average*.

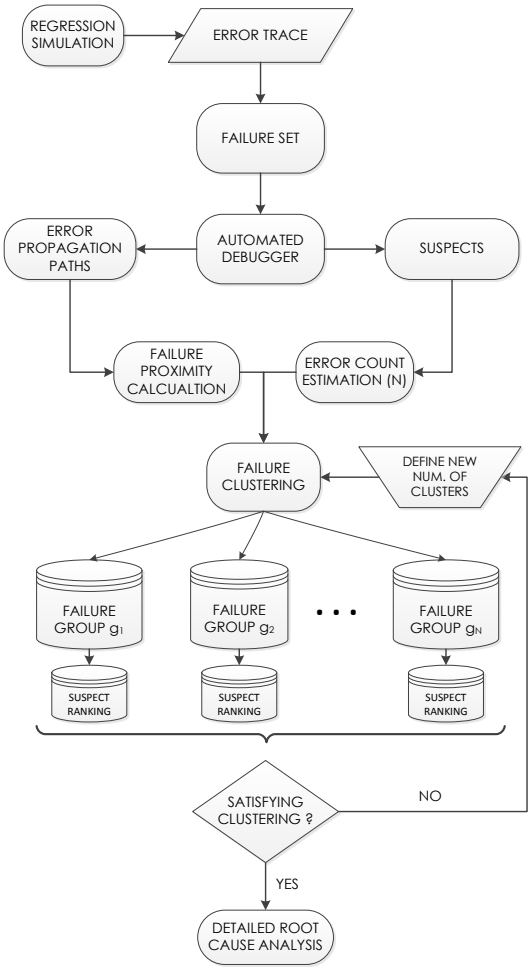


Fig. 5. Failure triage framework

D. Failure triage framework

The information that is embedded in the metrics described above, should be exploited for the last step of our framework which is the formation of groups of similar failures. To achieve this, we employ a hierarchical clustering algorithm [6].

Groups of failures are formed in a bottom-up fashion (agglomerative) by merging clusters that are likely to contain failures that are related. The merging stops when e (error count estimation) clusters are formed. Clustering algorithms use the notion of *distance* to define the relation between elements and clusters of elements. The smaller the distance is, the more related the elements are. In a clustering analysis context, the distance for a pair of observed failures $\mathcal{F}_i^j$, $\mathcal{F}_k^l$ is defined as:

$$dist(\mathcal{F}_i^j, \mathcal{F}_k^l) = 1 - \frac{D(\mathcal{F}_i^j, \mathcal{F}_k^l)}{W} \tag{4}$$

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 "round" clusters, which proved to be appropriate for our framework; similar failures tend to appear close to each other in a compact manner.

Fig. 4 illustrates an example of clustering for 10 distinct failures. Distances from Eq.4 between failures can be considered as Euclidean distances to define the position of each failure on a 2D plane [6]. By using Eq.3 we can constrain the number of clusters to be formed, so that they are equal to our estimation of how many errors should be considered. Since the frequencies of various suspects are already available to us, they can be provided as an output of the triage engine as well. Moreover, those frequencies can be sorted in order to generate a ranking for each suspect based on its appearance in the clustered failures. Therefore, for each group of failures the engineers that will pursue further root cause analysis, have both the error propagation paths and the suspect frequencies/ranking available. This valuable information can provide guidance for pin-pointing the actual error or can aid the engineer to identify possibly problematic groupings. The clustering algorithm can be executed with the error count estimation initially, but it depends on the engineer to figure out if this is an accurate assumption, and accordingly adjust the constraint and re-run the clustering algorithm. Based on the partial stages described above, Fig. 5 depicts the whole failure triage flow.

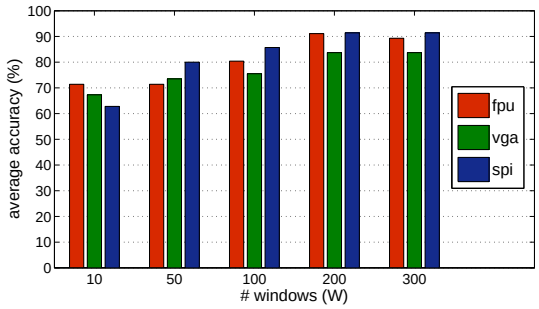
It should be clarified that the debugging step preceding the clustering process requires manual effort from the engineer(s). However, this does not add any overhead to the whole debugging flow. Rather, this manual task is simply moved earlier in the flow in order to generate the necessary signatures for failure clustering. As such, it will not have to be repeated once the groups of failures are passed to the rightful engineers.

IV. Experimental Results

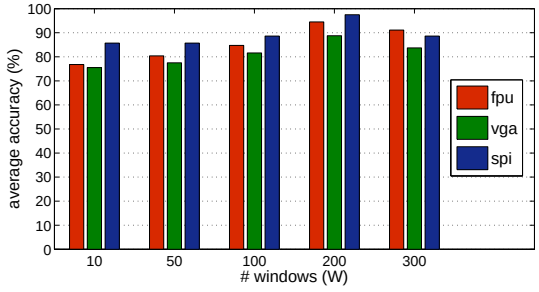
This section presents preliminary experimental results for the proposed failure triage framework. All experiments were conducted on a single core of an Intel Core i5 3.1 GHz workstation with 8GB of RAM. Three OpenCores [8] designs were used for the evaluation. The underlying automated debugging tool used for extracting the suspect locations and error propagation paths was implemented based on [4]. A platform coded in Python was developed to parse the returned results of the debugger, calculate the relevant metrics and perform hierarchical clustering on the resulting failures. For each design, a set of different errors was injected each time by modifying the RTL description. The injected RTL errors might correspond to a single signal or multiple signals. In the latter case, the module containing all the erroneous signals is considered as the error location when modules are selected as suspect locations. The prosposed framework does not add any new asserions or checkers to the designs. Rather, it exploits the existing ones. In total, twelve different regression simulations were run, generating a different number of failures each time, caused by a different set of errors in the design.

In order to evaluate how various parameters affect the accuracy of the resulting failure grouping, experiments were conducted by selecting different values for the number of windows described in Section III and the type of suspects (signals, modules). Fig. 6 illustrates accuracy results depending on the above parameters. The engine's accuracy is calculated by the ratio $\left(1 - \frac{misclassified\ failures}{total\ failures}\right)$. A misclassified failure is defined as a failure belonging to the wrong group. If a set of failures that are caused by the same error are wrongly split into two or more groups then only the group with the most failures is determined to contain correctly classified failures. The rest are considered misclassified.

From the observed results in Fig. 6, it can be concluded that accuracy rises to acceptable levels when internal signals are selected as suspects and error propagation paths are split into a relatively large number of windows. This can be justified as follows.



(a) Module suspect locations



(b) Signal suspect locations

Fig. 6. Effect of window number and suspect type on clustering accuracy

There are cases where more than one errors exist within the same design module. In those cases, the error count estimation will be rather optimistic and frequently generate less clusters compared to the actual number of design errors. Still, the group of failures that are caused by errors inside this module will be correct in most cases. Since the frequency of this module will be high in the suspect ranking for that group, it will aid the engineer to look for a error within the module; and this is usually acceptable for debugging purposes. However, we are interested in a more refined search. One that will provide the engineer with information regarding the actual number of errors within any module. Selecting internal signals as suspects provides a more detailed grouping, although it will sometimes generate a pessimistic error count estimation. This is due to noise included in Eq.2, because of the plethora of suspects that are mutual only for a subset of failures that should be put in the same group. However, the suspect ranking can help the engineer realise that two distinct groups should be merged together (i.e. if their distances are close and the same suspect appears high in the ranking for both groups). In that sense, failure triage can be manually re-executed by decrementing the number of generated clusters by one.

The number of windows also has a great impact on accuracy. Here, it is straightforward to realise that with a small number of windows it is hard to identify convergence and divergence of error paths, although their common propagation sub-paths will be determined. In other words, the smaller the number of windows is, the closer to a naive intersection of error paths this method moves. On the other extreme, the number of windows could be at most equal to the number of cycles of the shortest error propagation path whenever two paths are compared. Therefore, each window will include one cycle from one path and possibly multiple cycles from the other one. In that case, the way the error reaches before and propagates after every cycle is essentially discarded. This can potentially ignore valuable information or create very short decreasing and increasing subsequences in the calculation of Eq.1, that do not reflect to actual divergence and convergence of paths respectively.

Along these lines, the average accuracy ranges from 67.1%

TABLE I
PROPOSED FAILURE TRIAGE ENGINE PERFORMANCE

circuit	# gates	# errors	$\|\boldsymbol{F}\|$	e	accuracy			error rank (avg) ($N = e$)	time (sec)
					triage($N = e$)	*triage*(N = # errors)	*script*		
fpu	83303	4	**9** (15)	4	100%	100%	78%	1	12.7
		5	**12** (20)	6	83%	100%	67%	1	13.8
		6	**15** (24)	6	100%	100%	60%	1	16.4
		7	**20** (31)	7	95%	95%	65%	1.14	19.1
vga	72292	3	**8** (14)	4	88%	100%	75%	1	14.3
		4	**9** (15)	6	78%	89%	67%	1.17	15.2
		5	**13** (22)	5	100%	100%	54%	1	17.9
		6	**19** (29)	7	89%	95%	68%	1.14	18.8
spi	1724	2	**5** (8)	2	100%	100%	60%	1	12.0
		3	**8** (13)	3	100%	100%	63%	1	12.7
		4	**10** (16)	5	90%	100%	60%	1	13.4
		5	**12** (16)	5	100%	100%	67%	1	15.7
				AVG:	**93%**	**98%**	**65%**		**15.2**

with module suspects and 10 windows, up to 93% with signal suspects and 200 windows. An increase of 6.9% in accuracy is observed when switching from module suspects to signal suspects, for all window numbers. Likewise, as the window number increases, accuracy can improve up to 21.6% for module suspects and 14.3% for signal suspects. The minimum accuracy observed among all configurations was 61% for the *spi* design, for a testcase with 5 errors. The maximum number of windows for our experiments was 200, since the length of error propagation paths never exceeded 400 cycles. Notice that there is no benefit of arbitrarily increasing the number of windows. Rather, we observed a decrease of 3.1% in average accurcay when increasing the number of windows from 200 to 300. This drop in accuracy is in compliance with our observations presented in the previous paragraph.

Table I demonstrates detailed results for all twelve testcases and three designs. These results are generated with signals as suspects and 200 windows per error propagation path. The first and second columns refer to the design name and its size (in gates) respectively. The third and fourth columns contain the actual number of errors that were injected into the design and the number of total failures that the error trace caught. The fifth column 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 failure 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 failure triage flow when performed with our error count estimation or with the actual number of errors respectively. The ninth column presents the average rank of the suspect (in the suspect ranking list) that corresponds to the actual error location. The last column indicates the total time consumed by the calculation of the two metrics and the hierarchical clustering process.

It is ought to be mentioned that the actual number of failures caught by simulation regression might be significantly larger than the number presented in column 4 of Table I. However, a simple script (not the one we compare to) is used to identify trivial cases based on the error messages. As a result, multiple failures collapse into single ones, and those are used for our evaluations. In essence, we remove all trivial cases for our experiments, since the collapsed failures are manifestations of the same failure shifted in time. The above is a legitimately realistic approach when hundreds of failures occur during simulation. The actual numbers are given in parenthesis in column 4.

Based on Table I, the engine's average accuracy reaches 93% when the algorithm is executed with our initial guess (column 6) and may increase up to 98% when the engineer forces the number of clusters to be equal to the number of errors in the design. Generally, a good initial guess that reflects to the actual number of errors was observed in seven out of twelve testcases. In those cases, accuracy was 99% on average. On the other hand, in cases where the error count estimation is off by one or two clusters, accuracy drops to 86%. For the testcases in Table I, a conventional approach that employs scripts to perform failure grouping achieves an overall accuracy of 65% (i.e. one out of three failures are misclassified). In that context, the proposed method improves accuracy by 43% when the initial guess is utilized; a solid improvement that indicates the potential and applicability of the proposed framework. Moreover, the actual error is assigned a high rank in the suspect ranking list per cluster (column 9). Even if the highest rank is shared among suspects, it can significantly guide root cause analysis or aid the identification of bad clusters. Finally, computation of the two metrics and the clustering procedure consume an average of 15.2 seconds in total, which is acceptable for the purposes of failure triage.

V. CONCLUSION

In this work, a novel failure triage framework is proposed. The algorithm extracts signatures from debugging results to define relationship between various failures. Each signature is an approximation of the location of the root cause of a given failure. By determining mutuality between those signatures, failures that have a high probability of sharing the same root cause are grouped together. In order to quantify failure similarity and dissimilarity we introduce the concept of failure proximity and suggest a windowing scheme for its computation. Furthermore, we devise a speculative metric to estimate the number of co-existing errors. This also allows us to generate a ranking for the possible error locations to further guide root cause analysis and assign the grouped failures to the suitable engineer(s). The applicability and efficacy of the failure triage engine is demonstrated by experimental results within typical regression test flows, which indicate a significant increase in grouping accuracy compared to conventional automated methods.

REFERENCES

[1] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

[2] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

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

[4] 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.

[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] A. Jain and R. Dubes, "Algorithms for clustering data," 1998.

[7] G. J. Szekely 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.