Clustering-based Failure Triage for RTL Regression Debugging

Zissis Poulos[1], Andreas Veneris[1]
[1]Dept. of ECE, University of Toronto, Toronto, Canada.

Abstract

Regression verification at the pre-silicon stage has experienced a dramatic boost in capabilities over the past years. With the aid of assertions, improved simulation coverage and formal verification tools, a vast amount of trace data and myriads of failures are often generated after each regression run. Along these lines, modern flows face an emerging need to appropriately categorize, prioritize and distribute these failures to the engineer(s) best-suited for detailed debugging of each failure. This task is known as failure triage*. Despite its resource-intensive nature, triage remains a predominantly manual process. In this work, an automated data-mining failure triage framework is introduced that mines simulation and SAT-based design debugging data, uncovers relations among verification failures and automatically groups the related ones together. The core characteristic of the framework is a novel feature-based representation for verification failures and a new multiple-pass clustering strategy that surpass previous methodologies in accuracy, robustness and flexibility. The proposed triage engine achieves an 89% average accuracy in failure categorization and compared to existing solutions, it reduces the number of misplaced verification failures by 47% on the average.*

1 Introduction

Today, designs often undergo strenuous verification at the Register Transfer Level (RTL) to prevent functional errors from escaping to the tape-out stage where fixing costs can be daunting. Broadly speaking, verification can be either performed on-line or in regression mode, always with the use of simulation and formal verification tools that exercise circuit functionality.

Recent reports stress that verification poses a significant bottleneck in productivity by occupying up to 70% of the modern design cycle [1]. Further, they highlight design debug as the predominant bottleneck in verification flows, consuming up to 50% of the overall effort. However, debug in regression verification flows is bound by intrinsically different requirements and constraints compared to on-line verification. As such, the same surveys expose a significant gap in debugging automation capabilities between regression and on-line verification, in favor of the latter [1].

In on-line verification, debugging analyzes each failure in isolation, a process coined as *detailed debug*, to determine potentially erroneous design components that may be the cause of the observed failure. The cost associated with this process has been greatly alleviated by automated tools that employ powerful formal engines [2, 3, 4, 5].

On the other hand, debugging following regression is an inherently different process. When myriads of failures occur during regression tests, and before detailed debug commences, a fast coarse-grain pre-processing step has to be performed, known as *failure triage* [6, 7]. Failure triage, illustrated in Fig. 1, aims to group failures together and assign them to the appropriate engineers for further investigation. Roughly speaking, it consists of three tasks. First, high-level debug is performed to gain intuition about the approximate location of the error responsible for each exposed failure (i.e. design or testbench). Next, *failure binning* commences to group individual failures together that are suspected of originating from the same error source. The final triage task, called *failure bin distribution*, sends these bins to the best-suited engineer to perform detailed debug, identify the exact error location and eventually fix it [6].

Despite triage being declared as a fast growing regression problem [1, 6], current solutions rely on manual ad-hoc practices. Typically, engineering teams employ primitive forms of debug, such as simple error (*i.e.,* log) messages from end-to-end "golden-model" checkers, exception checkers and various assertions [7]. Due to the limited information conveyed by such means, failure relations are hard to identify, and therefore, the tasks of failure binning and bin distribution are often inaccurate. Evidently, such non-standard methods result into failures being constantly assigned and re-assigned to engineers until the rightful owner is found. As design and verification teams get more geographically dispersed, this is a process that may span hours or days consuming valuable resources.

Following the successful paradigm of data mining algorithms in the verification domain [8], engineers are now turning their attention to that field as a means to address the verification pain of failure triage. The work in [6] is the first to determine failure relations through heuristic-based metrics that combine information from

978-1-4799-4722-5/14/$31.00

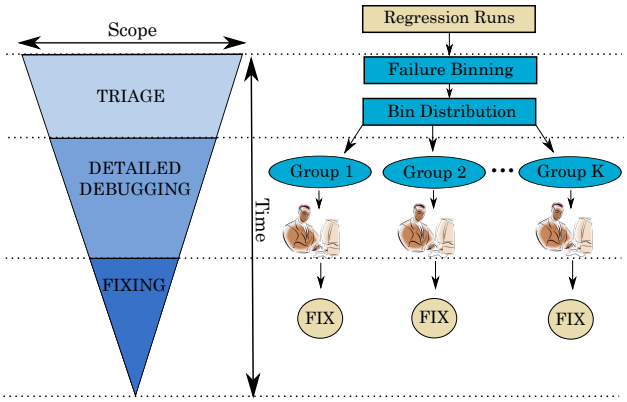


Figure 1: A modern design debugging flow

simulation and automated debugging. Failure binning and bin distribution are performed via a hierarchical clustering algorithm [9, 10]. However, the proposed method relies on greedy clustering algorithms that are sensitive to randomness and parameterization.

The work presented here introduces a novel and flexible automated failure triage engine that formulates the problem as a data mining clustering process and allows a variety of clustering algorithms to be applied [9]. It centers around three major contributions. First, it introduces a data weighting scheme based on simulation statistics to assign different levels of importance to data collected by SAT-based debugging. As a second contribution, it develops a feature-based representation to model verification failures as multi-dimensional objects. Based on this representation, failure relations can be easily computed as a distance measure and failure binning can be performed by a wide range of clustering algorithms. To emphasize these benefits, a clustering strategy that combines the merits of both connectivity-based and centroid-based clustering algorithms [9] is presented. As a last contribution, this paper shows that the aforementioned data weighting scheme can guide failure bin distribution, by determining how potentially erroneous design components should be prioritized for each group of failures.

The proposed triage engine is developed upon a traditional SAT-based debugging framework [3]. Experiments on four industrial designs show that it outperforms existing solutions with a 47% average reduction in the number of misplaced failures, which translates into an 89% average accuracy in failure binning. Further, it successfully guides bin distribution by identifying 98.6% of the injected errors as priority targets. Evidently these numbers demonstrate the robustness and effectiveness of the methodology in a modern regression verification environment.

The remainder of this paper is organized as follows. Section 2 offers notation and discusses prior work in failure triage for design debugging. Section 3 describes the proposed failure triage formulation and presents the clustering process. Finally, Section 4 discusses experimental results and Section 5 concludes the paper.

2 Preliminaries

2.1 Notation

Consider an erroneous design with a single or multiple errors in the RTL. When a mismatch between the expected "golden" value(s) (0,1 or X for unknown) and the observed one(s) is identified at some observation point (primary output, probed internal signal or the output of a property assertion), we say that a failure occurs. Suppose that the design undergoes regression verification that exposes N verification failures, denoted as $F_1, F_2, \ldots, F_N$. For each failure F_i there is a corresponding error trace E_i that contains the initial states and input vector sequence.

The output of a SAT-based debugger [3] for each error trace E_i is a set of design components (RTL blocks or signals) that can be responsible for failure F_i. This set is referred to as a *suspect* set for E_i denoted as $S_i = \{s_1, s_2, \ldots, s_{|S_i|}\}$ since each design component $s_j \in S_i$ can be modified to rectify the erroneous behavior exhibited by E_i. The length of an error trace E_i, denoted as $|E_i|$ is the number of cycles between the initial state (cycle 1) and the cycle where failure F_i is observed. Because of the exhaustive search performed by SAT-based debugging, the suspect set for each error trace E_i is an over-approximation of the actual error responsible for failure F_i [3]. That is, the design component where the actual error is located is guaranteed to be included in the suspect set.

Modern SAT-based mechanics allow debuggers to pinpoint the exact cycle where an erroneous value is excited at a suspect location to cause the observed failure [11]. That is, for each suspect set S_i there is an associated excitation set $T_i = \{t_1, t_2, \ldots, t_{|T_i|}\}$, where $t_j \in T_i$ is the excitation cycle for suspect $s_j \in S_i$. As an additional benefit, these tools return *error propagation paths* in the circuit that show how a value from a suspect location propagates through consecutive cycles to reach the failing output [11].

Example 1: *To demonstrate the above concepts consider an error trace, as depicted in Fig. 2. In that figure we show the sequential behavior of the circuit for that trace using its Iterative Logic Array (ILA) representation [3]. In more detail, an error at component s_2 is excited in cycle $m-2$ and propagates to cause a failure (F_1) at an observation point in cycle m. The generated error trace of length m is then passed to an automated debugger. The result is a suspect set $\mathcal{S}_1 = \{s_1, s_2, s_3\}$ of design components that can explain the wrong output. Suspects s_1, s_2 and s_3, excited in cycles k, $m-2$ and $m-1$ respectively, along with their propagation paths are illustrated in Fig. 2. Note that the erroneous component is included in the set S_1 as suspect s_2. For illustration purposes, suspects that correspond to the responsible design error are thereby shown by a solid circle (suspect s_2 in this case), whereas suspects that can explain the failure but are not actual erroneous components are shown by dotted circles (suspects s_1 and s_3 in this example).*

978-1-4799-4722-5/14/$31.00

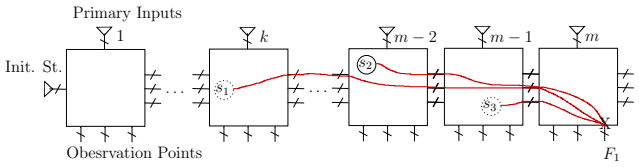


Figure 2: Error trace and suspect components

Simulation-based verification offers a wide range of metrics (code coverage, toggle coverage etc.) that can be exploited to extract useful information for the debugging process. Broadly speaking, knowing whether a design component is rigorously exercised or not, provides a measure of certainty when one attempts to estimate how reliably that component can be considered being error-free. In this work we focus on *toggle coverage* as a measure that provides such information. Toggle coverage (or simply toggling) refers to the number of times the logic value of specified nodes changes during simulation ($0/X \rightarrow 1$ or $1/X \rightarrow 0$). Among all design components, we are interested to extract this metric for the error suspects so we gain a better understanding for these critical components. In our presentation, we are interested to measure signal toggling within a specific time window for an error trace. Further, for each suspect these windows may differ with respect to various error traces.

Definition 1: *Given an error trace E_i, a* toggle window *with respect to suspect $s_j \in S_i$, is a pair of integers, denoted as ${W_j}^i = \langle p_j, q_j \rangle$, where p_j and q_j refer to the starting and end cycle between which toggling for suspect s_j is measured.*

In automated debugging, apart from internal signals, suspects may correspond to design modules as well [12]. To associate signal toggling with design modules the notion of toggling frequency is defined below.

Definition 2: *Given an error trace E_i, a suspect set S_i and a suspect component $s_j \in S_i$, we define the* toggling frequency *f_j^i of suspect s_j with respect to error trace E_i, to be the average toggling across all the input(s) of s_j within a specified window $W_j^i = \langle p_j, q_j \rangle$.*

In on-line verification, for each error trace the debugger is evoked and the engineer examines the resulting suspect set, which provides vital suggestions to track down the exact error location. The process then repeats for all available error traces. On the other hand, in regression verification engineers analyze multiple failures at a given time, as previously seen in Figure 1. For this strategy to be useful, each engineer should ideally be examining failures that are related and is familiar with. For that purpose, we define failure binning as follows.

Definition 3: *Given an erroneous design and a set of failures $F_1, F_2, \ldots, F_N$,* failure binning *is a complete disjoint partition of $F_1, F_2, \ldots, F_N$ into K groups (or bins) denoted as $G_1, G_2, \ldots, G_K$ such that each group contains failures that are likely to originate from the same error source.*

Based on the above definition, the relationship between failures that are binned together has to be clearly defined. Ideally, failures that belong to the same bin are all caused by the same design error. However, since state-of-the art debugging tools only approximate the actual error locations, it is practically impossible to develop a method that guarantees the above. Thus, it is paramount to define a measure of similarity between failures and group them accordingly.

Furthermore, a decision has to be made on the number of bins, K, to be formed by the engine. Ideally, K is equal to the number of co-existing errors responsible for the set of failures. However, in a realistic verification environment there is no prior knowledge on what this number is. A "guess" needs to be made and the quality of the triaging process should depend on how close this guess is to the number of injected errors [6].

Finally, there has to be a clear method for failure bin distribution. This decision is not always straightforward. A reasonable argument would be to parse suspect sets and assign failures to the engineers that are more familiar with these potentially erroneous design modules. Unfortunately, suspect sets are often large and may include components that are unrelated to the design error. Classical debugging techniques fail to guide this process because they are devoid of any sense of prioritization among suspect components.

2.2 Prior work

Conventional in-house triage techniques invest on scripts that parse error messages and in manual waveform analysis that often produce unreliable failure partitions. This is not surprising, because knowledge between failures and their culprit is limited at this stage.

The work in [6] proposes an automated failure triage process that overlaps with automated debug to leverage information conveyed by suspect sets. This is done by constructing metrics to heuristically quantify failure similarity. A suspect ranking scheme is also proposed that promotes locations potentially related to the design error against those that are likely unrelated. The generated similarity metric is used to perform failure binning via hierarchical clustering algorithms [9]. To estimate a reasonable value for K the authors propose an externally enforced stopping criterion for the clustering process, which is also based on suspect distribution among failures.

Although the above approach introduces clustering as a solution to triage, it faces some key limitations. First, suspect ranking is static, that is, conducted by simulation metrics that are measured within error trace windows of fixed length, although different parts of the error trace may include more useful information. The bottleneck here is that failure relations are solely represented by a heuristically computed real-valued similarity matrix, while the method is devoid of any feature-based failure model that treats them as separate objects. Clustering is, therefore, limited to a small class of connectivity-based algorithms. Finally, to estimate K, the failure clusters are assumed to be of similar size, an expectation that is not always realistic.

978-1-4799-4722-5/14/$31.00

3 Failure Triage Engine

In this section we propose a complete failure triage engine that follows a dynamic weighting approach for mined simulation and debugging data. At the core of the engine lies a novel feature-based failure model that allows a combination of clustering algorithms to be applied for the purposes of failure binning.

3.1 Data Collection

As discussed in Section 2, traditional failure binning and bin distribution is generally performed with very high uncertainty, because error logs and human insight offer empirical and often unreliable estimations. Suspect sets, on the other hand, are the result of a formal exhaustive search. Thus, collecting and leveraging these sets can provide the means of expressing failure similarity based on more accurate data.

To this end, once regression finishes with a set of failures $F_1, F_2, \ldots, F_N$ and corresponding error traces $E_1, E_2, \ldots, E_N$, a baseline SAT-based debug step for each error trace is performed to collect suspect sets $S_1, S_2, \ldots, S_N$ and respective excitation sets $T_1, T_2, \ldots, T_N$. Let $\{s_1, s_2, \ldots, s_M\}$ be the set of all distinct suspect locations in $\bigcup_{i=1}^{N} S_i$. Note that a suspect location s_j may belong to more than one suspect sets, and we say that s_i is shared among these sets, or equivalently, among those traces/failures. A shared suspect may have a different excitation cycle in each suspect set it appears in. For each suspect s_j there is a set of excitation cycles as observed in sets $T_1, T_2, \ldots, T_N$, denoted as $T_{s_j} = \{t_j^1, t_j^2, \ldots, t_j^N\}$, where t_j^i is the excitation cycle of suspect s_j as it is observed in excitation set T_i, and thus in error trace E_i. If s_j is not a suspect for E_i then $t_j^i = 0$.

3.2 Data Weighting Scheme

In any case, each suspect set S_i provides some guidance to the general error location related to failure F_i. However, formal debuggers, including SAT-based ones, may return very large suspect sets for each exposed failure. Some of the suspect locations (i.e. reset signals, primary inputs, dangling logic, bit-flips etc.) explain the failure but may be irrelevant to the erroneous module or signal responsible for it. These suspects are said to bear noise and should be assessed as less significant. Thus, all collected suspect components need to be appropriately weighted to quantify their significance in every error trace that they appear.

Ideally we want to identify and promote suspects that exhibit behavior similar to that of typical design errors. This is because such locations generally form better candidates when attempting to determine the culprit of a failure. Recent work has experimentally shown that there are two properties often observed in such suspect components. The first is temporal proximity to the observed failure [13]. That is, typical design errors are expected to be excited only a few cycles before the failure is observed, since they can quickly propagate to observation points in most cases. Second, these locations are excepted to exhibit low toggling frequency measured from the initial state up to the cycle where the suspect is excited [6]. The argument behind the latter is that typical RTL errors are relatively "easy" to excite in the majority of cases.

To address this, we propose a suspect weighting scheme that is based on the above two criteria. Thus, we promote suspect locations that exhibit both temporal proximity and small toggling frequency, and penalize them otherwise. Precisely, we expect that a low suspect toggling frequency measured between distinct excitations of that location indicates that the corresponding error is relatively easy to excite, and vice versa.

Consider, for example, a suspect location s_j for error trace E_i, with respective excitation cycle t_j^i. Assume that the same suspect location s_j also appears in the suspect set of some other trace E_k, where its excitation cycle t_j^k lies before cycle t_j^i. Then s_j should be considered easy to excite, if its toggling frequency f_j^i is low between these cycles. This is because, starting from cycle t_j^k, the inputs of the suspect component need to toggle only a small number of times until the next excitation is observed in cycle t_j^i. Therefore, it is reasonable to measure frequency f_j^i within a window that starts at the preceding suspect excitation cycle, t_j^k.

To do so, we define $\hat{t}_j^1, \hat{t}_j^2, \ldots, \hat{t}_j^N$ to be the excitation sequence for suspect s_j, which is generated by sorting the excitation cycles in T_{s_j} in increasing order. The toggling frequency f_j^i of suspect s_j with respect to E_i is thereby measured in window:

$$W_j^i = \begin{cases} \langle 1, t_j^i \rangle & , t_j^i = \hat{t}_j^1 \\ \langle t_j^k, t_j^i \rangle & , otherwise \end{cases} \quad (1)$$

where $t_j^k = \hat{t}_j^{l-1}$, if $t_j^i = \hat{t}_j^l$, for some $\hat{t}_j^l$ in the excitation sequence of suspect s_j.

Observe that according to Eq. 1, if suspect location s_j is unique to trace E_i then its window begins at cycle 1, since $|T_{s_j}| = 1$ and, hence, $t_j^i = \hat{t}_j^1$. If s_j appears in multiple traces but has its first excitation observed in E_i, then its window again begins at cycle 1, since t_j^i is the earliest excitation ($t_j^i = \hat{t}_j^1$). In any other case, frequency is measured within a window that begins at the previous suspect excitation.

Example 2: *Suppose that two failures F_1 and F_2 are exposed after regression, with error traces E_1 and E_2, respectively, as shown in Fig. 3. For failure F_1 three suspects are extracted, namely s_1, s_2 and s_3, whereas for failure F_2 the debugger returns s_1 and s_2. Suspects s_1 and s_2 are shared among F_1 and F_2, for a total of three distinct suspect locations. Among all possible trace windows, two are shown as an example*

978-1-4799-4722-5/14/$31.00

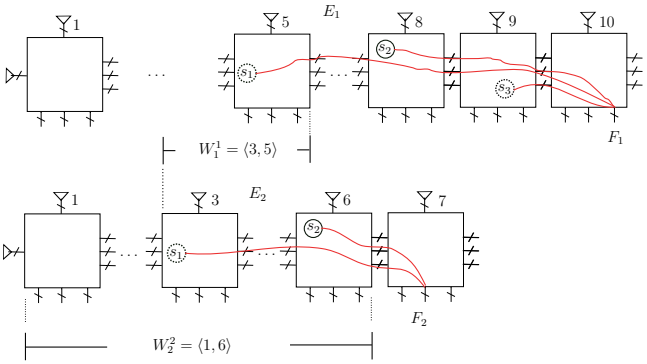


Figure 3: Error trace for toggling frequency measurement

in the figure. Window $W_1^1 = \langle 3, 5 \rangle$ spans cycles 3 to 5 and the frequency of suspect s_1 with respect to trace E_1 is measured within W_1^1. The window starts at cycle 3, because s_1 has its previous excitation observed in cycle 3, which is captured by trace E_2. Similarly, the toggling frequency of suspect s_2 with respect to error trace E_2 is measured within window $W_1^1 = \langle 1, 6 \rangle$, since it is first excited in cycle 6, captured by the same trace.

The second criterion that promotes temporal proximity of suspect s_j to the failing observation point can be expressed by the number of cycles between the excitation cycle t_j^i and the cycle where failure F_i is observed. Since each error trace E_i begins at cycle 1, the failure is observed at cycle $|E_i|$. Thus, for suspect s_j, temporal proximity with respect to F_i is quantified by computing $|E_i| - t_j^i$.

Once toggling frequency and temporal proximity are calculated for all suspects $s_1, s_2, \ldots, s_M$, a weight, denoted as w_j^i, assigns various levels of significance to each suspect location s_j with respect to failure F_i. The weight w_j^i is given as follows:

$$w_j^i = \left(1 - \frac{|E_i| - t_j^i}{|E_i|}\right) + \left(1 - \frac{f_j^i}{\max\limits_{s_k \in S_i} f_k^i}\right) \qquad (2)$$

In Eq. 2, the first term promotes the significance (weight) of suspect s_j with respect to failure F_i when the suspect excitation cycle is observed close to the cycle where the failure is exposed, and penalizes it otherwise. On the other hand, the second term penalizes high frequencies, thus reducing the score of suspects that are hard to excite, which based on our assumptions are considered "noisy". Note that in the second term, the denominator $\max\limits_{s_k \in S_i} f_k^i$ is used to normalize over the maximum suspect toggling frequency observed for failure F_i. The overall score is the quantified summation of both criteria that are discussed in this work.

3.3 Feature-based Failure Representation

As shown in Section 2, failure binning is defined as a complete disjoint partition of failures $F_1, F_2, \ldots, F_N$, which can be naturally formulated as a clustering problem, since failures constitute unlabeled objects. In order to form bins (clusters) of related objects, the similarity or dissimilarity between each pair of failures needs to be determined.

To achieve this, we first introduce a feature-based representation for each failure. Then we map failures to data points into a high-dimensional Euclidean space where pairwise failure dissimilarity can be naturally expressed by the Euclidean distance between the corresponding data points [9].

Our goal when constructing a feature-based failure representation is to express each failure F_i in terms of its suspect locations and their significance. Precisely, we associate each failure F_i to a feature vector, denoted as $\vec{F_i} = [x_1^i, x_2^i, \ldots, x_M^i]$, where:

$$a_{ij} = \begin{cases} w_j^i & , s_j \in S_i \\ 0 & , s_j \notin S_i \end{cases} \qquad (3)$$

is a variable taking a value equal to the weight of suspect s_j with respect to failure F_i or a value of 0 if s_j does not appear in the suspect set for failure F_i. With this model, each feature encodes the existence (or absence) of specific suspect locations and their corresponding significance, which is computed through the weighting scheme.

The above model allows us to generate a data matrix, denoted as $D_{N,M}$, and given as:

$$D_{N,M} = \begin{pmatrix} x_1^1 & x_2^1 & \cdots & x_M^1 \\ x_1^2 & x_2^2 & \cdots & x_M^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^N & x_2^N & \cdots & x_M^N \end{pmatrix} \qquad (4)$$

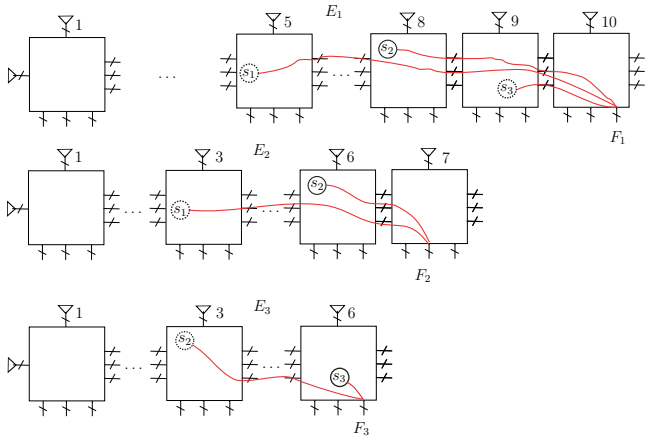
In matrix $D_{N,M}$, each row i corresponds to failure F_i and each column j corresponds to the contribution of suspect s_j with respect to each failure. By using the data matrix, each failure F_i can be mapped to an individual data point into an M-dimensional Euclidean space. Then, dissimilarity between two failures F_i and F_j can be expressed as their Euclidean distance, denoted as d_{ij} and given as:

$$d_{ij} = ||\vec{F_i} - \vec{F_j}|| \qquad (5)$$

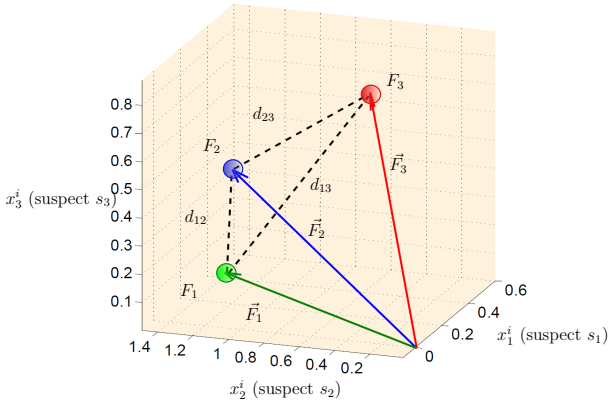
Pairwise distance for the whole data set is encapsulated by a distance matrix, denoted $d_{N,N}$, which is given as follows:

$$d_{N,N} = \begin{pmatrix} d_{11} & d_{12} & \cdots & d_{1N} \\ d_{21} & d_{22} & \cdots & d_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N1} & d_{N2} & \cdots & d_{NN} \end{pmatrix} \qquad (6)$$

978-1-4799-4722-5/14/$31.00



(a) Failures F_1, F_2, F_3, and corresponding suspect locations



(b) Failures F_1, F_2, F_3 mapped to a 3-dimensional space

Figure 4: Feature-based failure representation

Example 3: *Consider three failures F_1, F_2 and F_3 exposed after regression, with error traces E_1, E_2 and E_3, respectively, as shown in Fig. 4(a). In total three distinct suspect locations are observed, s_1, s_2 and s_3. These suspect components along with their scores per error trace, form the base of the proposed failure representation. The generated feature vectors for each failure become $\vec{F_1} = [w_1^1, w_2^1, w_3^1]$, $\vec{F_2} = [w_1^2, w_2^2, 0]$, $\vec{F_3} = [0, w_2^3, w_3^3]$, by assigning each suspect weight w_j^i to the corresponding random variable x_j^i, according to Eq. 3. In this example three distinct features are used, thus the failures are mapped to a 3-dimensional Euclidean space, as shown in Fig. 4(b). The Euclidean distances d_{12}, d_{13}, d_{23}, are computed based on Eq. 5.*

Generally, we expect a small distance to indicate close relation and vice versa. Intuitively, failures that share many suspects in common with similar weights are expected to appear close to each other. The absence of shared suspects between two failures and/or large variations in suspect weights indicate a weaker relation and these failures are mapped to data points that are relatively distant.

Observe that, in the example of Fig. 4, failure F_1 and F_2 refer to different observation points, but are nonetheless caused by the same design error, which is returned as suspect s_2 in both traces. The existence of this suspect location along with possibly other locations related to the design error, place these failures relatively close in the Euclidean space. As such, a relation is exposed that is otherwise hard to identify by simply observing the failing outputs, which is conventionally done by error log parsing or failing output monitoring.

3.4 Failure Binning and Bin Distribution

In order to perform failure binning, we desire a partition of failures $F_1, F_2, \ldots, F_N$ into K clusters of related failures. Although relation can be derived by pairwise distance, a major bottleneck in clustering quality is that K is not known a priori. To ease this problem, we decide to perform failure binning through a two-pass clustering process.

The first pass produces coarse nested partitions of the data set and is performed through agglomerative hierarchical clustering. The process is often represented and visualized by a dendrogram, as shown in Fig.5, for a real data sample of 29 failures. The algorithm first generates a trivial partition consisting of N clusters where each failure F_i is its own singleton cluster. This initial partition corresponds to the bottom tree level, as in Fig.5. Then it proceeds by iteratively merging the two closest clusters until a single cluster entailing all failures is produced (root of the tree). At the end of the process all intermediate partitions involving 1 to N clusters are available. The benefit of this method is that it does not require us to specify K beforehand.

The decision to merge two clusters is based on a linkage criterion that determines how the distance between two clusters is defined and, at each iteration, merges the pair with minimum cluster distance. In Fig. 5 cluster distance (or merge cost) corresponds to the height of each tree link that connects two clusters at each iteration. Among a wide range of linkage criteria we select Ward's method where at each step we merge the pair of clusters that lead to minimum within-cluster variance after merging [10]. More precisely, Ward's Method says that the merge cost between two clusters A and B, denoted as Δ_{AB}, is the amount of increase in variance of the data points that belong to these clusters, if we

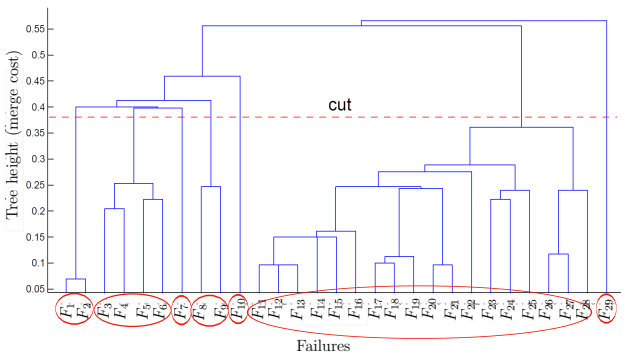


Figure 5: Hierarchical clustering dendrogram

978-1-4799-4722-5/14/$31.00 ©2014 IEEE

decide to merge them into a larger one. Formally:

$$\Delta_{AB} = \sum_{F_i \in A \cup B} ||\vec{F_i} - \vec{m}_{A\cup B}||^2 - \sum_{F_i \in A} ||\vec{F_i} - \vec{m}_A||^2 - \\ - \sum_{F_i \in B} ||\vec{F_i} - \vec{m}_B||^2 \quad (7)$$

where $\vec{m}_k$ is the center (mean) of cluster k.

In this work we observe that failures tend to form compact clusters and Ward's method promotes this by keeping the increase in sum of squares as small as possible after each intermediate step.

Among all generated partitions we aim to select the one that involves the most reasonable number of clusters, K. In order to determine a good value for K we identify the iteration that incurs a large merge cost, as given in Eq.7, relative to the average merge cost across all iterations. Intuitively, this iteration attempts to merge clusters that are "unnaturally" distant and often indicates the step where a bad partition is about to be generated. As such, once this step is determined we keep the partition that precedes the iteration. This decision essentially corresponds to cutting the tree at the respective link height. All failures that are linked in heights below the cut are placed in the same group, as shown in Fig.5, where a partition of 7 distinct failure groups is selected.

Although this process can quickly guide the selection of K, the returned partition can be further refined. This is because the linkage criterion makes a greedy selection at each iteration and the sum of squares is often far from any local minima.

To produce a finer partition, we perform a second clustering pass, based on K-means [9]. The algorithm requires K to be specified a priori and, in our case, the value comes from the partition that is selected in the first pass. K-means clustering minimizes the within-cluster sum of squares and converges to a local minimum, but traditionally requires multiple initializations to assign cluster centers (means) for a given K. In our case, we deal with this by assigning the cluster centers to be the ones of the partition that is selected after the first pass.

After the second clustering pass, a partition consisting of failure bins $G_1, G_2, \ldots, G_K$ is available, and the task of failure bin distribution takes place. We implement this process by generating a ranked set of shared suspects among failures that belong to each failure bin. Precisely, for each group G_i we first compute $S_{G_i} = \bigcup_{F_j \in g_i} S_j$, which is the set of distinct suspects among failures within group G_i. Then, for each set S_{G_i} we order the suspect components in decreasing order of average weight (Eq. 2) across all failures in G_i. This process produces a ranked version of S_{G_i}, denoted $S^R_{G_i}$.

By examining the ranked set $S^R_{G_i}$ for a failure bin G_i we can determine the suspect locations that have been assigned high scores for that particular set of failures. These suspects are considered of higher significance and should be prioritized when performing detailed debug for failures in bin G_i. Therefore we can guide the bin distribution process by assigning bin G_i to the engineer responsible for the design module(s) that correspond(s) to the high ranked suspects in $S^R_{G_i}$.

4 Experimental Results

This Section presents experimental results for the proposed triage framework. All experiments are conducted on a single core of an Intel Core i5 3.1 GHz workstation with 8GB of RAM. Four *OpenCores* [14] designs are used for the evaluation (`vga`, `fpu`, `spi` and `mem_ctrl`). The underlying automated debugging tool used for extracting the suspect locations is implemented based on [3]. A platform coded in Python is developed to parse debugging and simulation results, calculate suspect weights and perform the clustering process on the generated failures. For each design, a set of different errors is injected each time by modifying the RTL description. The types of the injected RTL errors are not generated randomly. Rather, they resemble typical human-introduced errors (missing pipeline stages, incorrect read pointers, bad stimulus etc.) that lead to non-trivial triage scenarios. In total, twenty regression simulations are run, generating a different number of failures each time, caused by a different set of errors.

For each design, a pre-generated set of test sequences is used that is stored in vector files. Each regression run involves hundreds to thousands of input vectors. For the purpose of capturing failures we use end-to-end checkers that compare the expected value for various operations, exception checkers and various assertions throughout the designs.

Table 1 summarizes benchmark information and statistics per regression run. From left to right, columns show the circuit name and number of gates, an enumeration for regression runs, the number of input vectors, the number of simultaneously injected RTL errors, the number of exposed failures (N), and finally

Ckt. (# gates)	Test No.	# vectors	# errors	# fail. (N)	# susp. (M)
vga (72292)	1	25206	4	45	36
	2	25206	7	62	40
	3	25206	8	97	61
	4	31870	10	106	129
	5	31870	13	121	155
fpu (83303)	6	17365	3	19	28
	7	17365	7	30	29
	8	20094	7	55	74
	9	41759	9	83	60
	10	41759	11	125	111
spi (1724)	11	4573	3	13	38
	12	4573	5	28	46
	13	4573	6	51	82
	14	5019	8	54	79
	15	5019	9	72	113
mem_ctrl (46767)	16	10834	3	17	24
	17	10834	5	32	45
	18	10834	7	31	29
	19	13370	8	66	94
	20	13370	11	95	137

Table 1: Benchmarks and Regression Statistics

978-1-4799-4722-5/14/$31.00

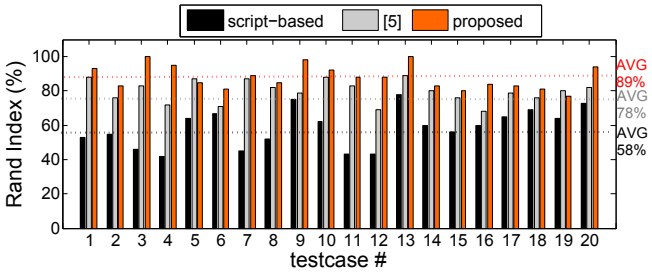


Figure 6: Engine accuracy vs. existing methods

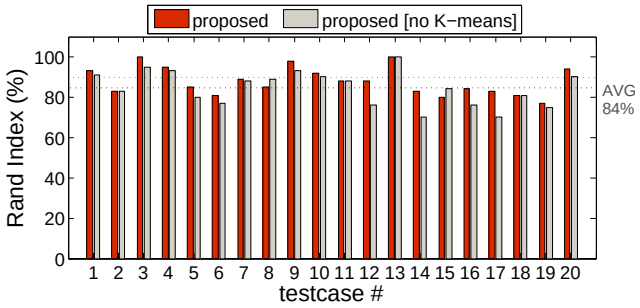


Figure 7: Impact of K-means in clustering accuracy

the number of distinct suspect components (M) generated by SAT-based debugging per regression run. The SAT-based debugging tool used in our flow returns suspect components in a hierarchical fashion, from the module level down to the signal level [3, 12]. When collecting suspect locations, we favor suspects that correspond to design modules and/or Verilog/VHDL blocks, rather than suspects at the signal/gate level. This is because we aim to keep the number of dimensions (M) relatively manageable and avoid critical impediments by the "curse of dimensionality" [9].

The success of a failure binning algorithm may be naturally quantified by two factors. The quality of the guess on the number of injected errors that essentially determines the number of bins, and the ratio of correct clustering decisions (clustering accuracy) when failures are grouped together.

In order to evaluate clustering accuracy, we use a metric called `Rand Index` (`R.I.`) [9]. This metric compares the estimated clustering against a reference failure binning, with the latter corresponding to an ideal partition where all failures are grouped with 100% accuracy. The metric ranges from 0 to 1 and represents the fraction of correct clustering decisions. Clustering accuracy is hence computed as $100 \times$ `R.I.`%.

Fig. 6 compares clustering accuracy between the proposed framework, the approach described in [6], and an in-house script-based technique. The triage engine proposed in this work outperforms the framework in [6] in 18 out of 20 regression runs and achieves much higher accuracy in all regression runs compared to the script-based method. Across all regression runs, the proposed engine achieves 89% clustering accuracy on average, compared to the 78% and 58% accuracy of [6] and the

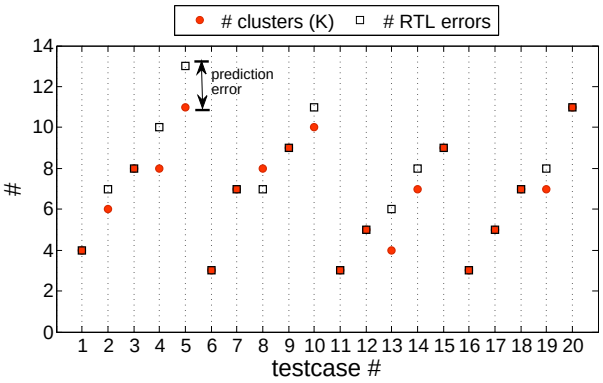


Figure 8: Predicted K vs. number of injected RTL errors

script-based binning, respectively. This translates into a 47% average reduction in clustering error rate compared to the existing automated method in [6].

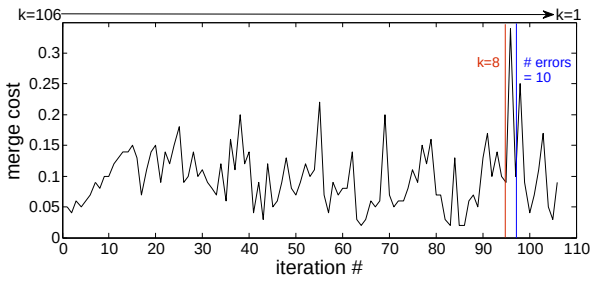
In Fig. 7 we explore the effect of the K-means clustering pass on the engine's accuracy, by comparing clustering quality between the two-pass strategy (`proposed`) and the case where we choose to omit the K-means pass (`proposed [no K-means]`). We observe that refining the initial hierarchical partition through K-means improves overall accuracy in 12 out of 20 testacases, whereas in 6 testcases no improvements appear. Promoting compact failure clusters and small sum of squares experimentally appears to be the right decision, since both hierarchical clustering in isolation and in combination with K-means achieve generally high accuracy. However, K-means further reduces the sum of squares across the partition and this may lead to the creation of some bad clusters in the case where some failures are spread away from the cluster center. As such, in 2 out of 20 testcases, we observe that the K-means pass harms clustering quality. Nevertheless, using K-means improves triage accuracy by 5.9%.

After the hierarchical pass, we perform a tree cut to

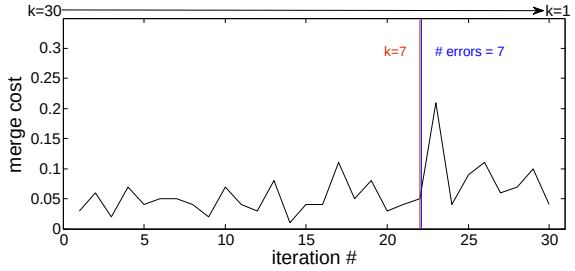
Test No.	$\lvert S^R_{G_i} \rvert$ (avg.)	error rank		
		high	low	mean
1	16	1	11	4
2	11	1	6	3
3	15	2	7	5
4	29	1	15	5
5	26	1	14	4
6	17	2	7	5
7	12	1	12	4
8	20	2	9	3
9	13	1	8	2
10	21	2	7	3
11	18	1	14	6
12	17	1	7	3
13	17	1	5	2
14	13	2	13	7
15	16	1	10	4
16	10	1	4	1
17	13	1	5	2
18	12	1	5	1
19	18	2	6	3
20	21	1	6	2

Table 2: Suspect ranking

978-1-4799-4722-5/14/$31.00



(a) testcase No. 4 (vga)



(b) testcase No. 7 (fpu)

Figure 9: Hierarchical clustering merge cost and tree cut

select a reasonable partition, and we do so by cutting at the height where the merge cost is 3 to 5 sigmas above the mean. To illustrate how our prediction performs, Fig. 8 shows the difference between the number of injected RTL errors and the number of clusters that are selected from the dendrogram cut process after the first hierarchical pass. We observe that in 12 regression runs the prediction is perfect, and this has a great impact in accuracy. As show in Fig. 6, for these 12 regression runs triage accuracy reaches an average of 94%. Still, there are 7 testcases where the prediction fails to capture the actual number of errors and generates 1-2 less clusters that required. This is an expected effect of the "curse of dimensionality", since there are various clusters of extreme density where failures appear in such spacial proximity that any sub-clusters are always merged with small cost even if they should ideally remain separated. Interestingly, we predict a larger number of clusters than needed only in a single regression run (testcase 8, 1 extra cluster).

In Fig. 9(a) and Fig. 9(b) we show how the merge cost evolves in time during the hierarchical clustering step for testcases 4 and 7, respectively. In the former, the merge cost where we pick to perform a tree cut is slightly higher compared to the merge cost that corresponds to the number of injected errors. As such, by cutting higher in the hierarchy, we predict two less clusters compared to the number of errors. In testcase 7, on the other hand, the merge costs agree, and we correctly predict 7 clusters. Experimentally, cutting at the tree height where the merge cost is 3 to 5 sigmas above the mean performs acceptably. This is because our feature-based failure representation creates well separable clusters in the majority of cases.

To measure how efficiently the framework guides the

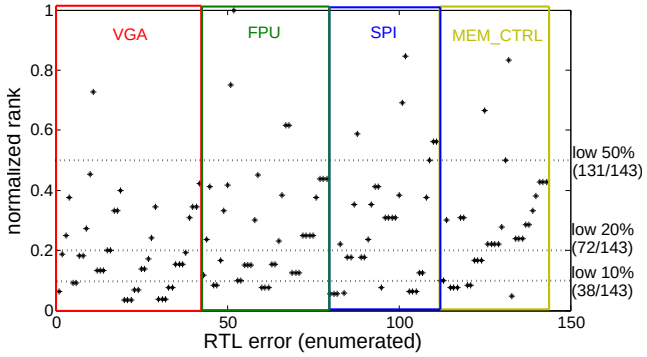


Figure 10: Normalized ranks for RTL errors

process of failure bin distribution, we examine the rank that is assigned to each RTL error in the group of failures where it appears as a suspect. Recall that for each failure bin G_i we compute a set $S^R_{G_i}$ of ranked suspects, where a high in rank suspect component is assumed of high importance for that given failure bin. Ideally we want the error responsible for bin G_i to appear high in rank, so that it becomes a priority location of detailed debugging.

Along these lines, Table 2 shows how ranks are allocated to RTL errors per regression run. Precisely, the first and second columns respectively include the regression number and the average size of ranked suspect sets across all failure bins for each regression run. The third, fourth and last columns show the highest, lowest and mean rank that is assigned to RTL errors across all failure bins for a particular regression run. Note that a rank of 1 indicates that the RTL error has the maximum average score within the bin that it appears (highest rank). Fig. 10 shows in greater detail the allocation of ranks for each design. Ranks are normalized over the ranked suspect set size. We observe that 26.6% of the errors appear in the upper 10% of the ranked sets, half of the errors appear in the upper 20%, and generally the vast majority (98.6%) appear in the upper half of the ranked sets. Results in Table 2 and Fig. 10 demonstrate that the data weighting scheme assigns scores that tend to push the injected RTL errors higher in rank, as desired. During experimentation we observe that injected errors that have a minor effect to the design by causing only a few failures and/or by being hard to excite are generally the ones that the engine fails to promote successfully.

Finally, we measure time consumption for the triage engine and present a break-down of the total time based on the three major steps: the baseline SAT-based debugging session, the data collection and weighting step and the clustering pass. Results are shown in Fig. 11. SAT-based debugging is an exhaustive search process and, as expected, dominates the total time (91% on average), with data collection and weighting following (7% on average). Clustering accounts for only 2% of the total time, on average. Data collection and weighting has a complexity of $O(N \times M \times L)$ where

978-1-4799-4722-5/14/$31.00 ©2014 IEEE

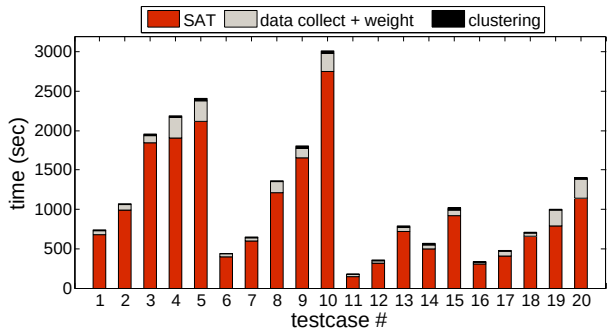


Figure 11: Triage time consumption break-down

N is the number of failures, M the number of distinct suspects and L the maximum error trace length for a particular regression run. The clustering process has a complexity of $O(N^{M\times K}\log N)$, dominated by K-means clustering.

It should be clarified that the baseline SAT-based debugging session is always performed whether triage is present or not in the flow. As such, the proposed triage framework does not incur the time overhead related to this task. It simply requires the baseline debug session to be moved one step higher in the flow, before failure binning and bin distribution take place.

5 Conclusion and Future Work

To summarize, this work introduces a novel solution to tackle the growing problem of failure triage in design debugging. It proposes a framework that addresses the critical tasks of failure binning and failure bin distribution by formulating triage as a machine learning clustering process and proposing novel metrics to express verification failures with a compact and feature-based representation model. Preliminary results show that this framework surpasses existing methodologies in clustering accuracy.

Failure triage is a task that inherently involves uncertainty and there will always be fertile ground for improvements. These may come in various forms, such as more accurate statistical models to rank and de-noise suspect sets and methods to exploit information from passing tests during regression. Moreover, one can explore failure representations that involve suspects at a much higher granularity, while using dimensionality reduction for the resulting high-dimensional data. Finally, it would be interesting to explore the efficacy of a wide range of sophisticated clustering strategies and algorithms that can exploit the flexibility of the feature-based failure representation that is proposed here.

References

[1] H.Foster, "From volume to velocity: The transforming landscape in function verification." in *Design Verification Conf.*, 2011.

[2] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita, "A formal approach for debugging arithmetic circuits," in *IEEE Transactions on CAD*, vol. 28, no. 5, May 2009, pp. 742–754.

[3] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Transactions on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[4] S. Mirzaeian, F. Zheng, and K. Cheng, "Rtl error diagnosis using a word-level sat-solver," in *International Test Conference*, 2008, pp. 1–8.

[5] K. hui Chang, I. Wagner, V. Bertacco, and I. L. Markov, "Automatic error diagnosis and correction for rtl designs," in *Proc. International High Level Design Validation and Test Workshop (HLDVT) 2007*, pp. 65–72.

[6] Z. Poulos, Y. Yang, and A. Veneris, "Simulation and satisifiability guided counter-example triage for rtl design debugging," in *Int'l Symposium on Quality Electronic Design*, 2014, pp. 394–399.

[7] S.Safarpour, B.Keng, Y.S.Yang, and E.Qin, "Failure triage: The neglected debugging problem," in *Design and Verification Conference*, 2012.

[8] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Design, Automation and Test in Europe*, 2010, pp. 626–629.

[9] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2007.

[10] 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.

[11] B. Keng and A. Veneris, "Path directed abstraction and refinement in sat-based design debugging," in *Design Automation Conf.*, 2012.

[12] M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir, "Debugging sequential circuits using Boolean satisfiability," in *Int'l Conf. on Computer Aided Design*, 2004, pp. 204–209.

[13] S. Safarpour, A. Veneris, and F. Najm, "Managing verification error traces with bounded model debugging," in *ASP Design Automation Conf.*, 2010.

[14] OpenCores.org, "http://www.opencores.org," 2007.