# Suspect Set Prediction in RTL Bug Hunting

Neil Veira, Zissis Poulos and Andreas Veneris
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
Email: {nveira, zpoulos, veneris}@eecg.toronto.edu

***Abstract*—We propose a framework for predicting erroneous design components from partially observed solution sets that are found through automated debugging tools. The proposed method involves learning design component dependencies by using historical debugging data and representing these dependencies by means of a probabilistic graph. Using this representation, one can run a debugging tool non-exhaustively, obtain a partial set of potentially erroneous components and then predict the remaining by applying a cost-effective belief propagation pass. The method can reduce debugging runtime when it comes to multiple debugging sessions by 15x on the average while achieving a 91% average prediction accuracy.**

## I. Introduction

When functional verification captures a mismatch between the implementation of a design and its specification, it returns an error-trace, comprising the sequence of input stimuli and state transitions that lead to the discrepancy. The task of design debugging pertains to analyzing each error trace to localize the design error responsible for that behavior. Today, debugging largely remains a manual task in a typical design cycle. It often consumes up to 60% of the total verification effort [1], as the increasing size and complexity of designs intensifies the challenge of bug localization.

To reduce debugging costs that jeopardize time-to-market, several efforts in Computer Aided Design (CAD) have been made to automate the process. Given an error trace, a typical automated debugger outputs a set of potentially buggy design locations (RTL lines or blocks), referred to as the *suspect set*. Each suspect points to a design location where a modification can rectify the failure. Most state-of-the-art debuggers perform multiple calls to formal engines, such as Boolean Satisfiability (SAT), Quantified Boolean Formulas, and Maximum Satisfiability [2]–[5], until all suspects are exhaustively found. The computational cost and memory footprint this incurs remains an impediment for these tools to scale to larger designs.

The problem is especially profound when considering high-level debugging tasks, such as failure triage and/or debugging in regression mode. In triage the goal is to identify which error traces are likely to originate from the same design module before doing detailed debugging. Recent triage methods run these formal tools for each error trace to extract an over-approximation of the high-level error location, in order to quantify error trace similarity [6]. Thus, for a decision to be made at the triage stage all error traces need to go through an exhaustive formal tool process, which increases the computational cost. Moreover, debugging in regression mode often involves multiple errors being placed in the queue for analysis and fixing, which again requires running these formal tools multiple times.

In both these scenarios, there is an underlying redundancy that can be exploited to reduce the total cost. Particularly, for the set of all design components there generally exists a partitioning into *suspect classes*, whereby if a component appears in a suspect set for a particular error trace then all other components in the same suspect class will also be in the suspect set. Of course, exactly computing these classes is intractable. Methods that employ structural dominance [7] and incremental SAT [8] attempt to approximate them, but still require formal subroutines to do so. This work conjectures that given sufficient historical debugging data, one can *probabilistically* learn design component dependencies (i.e., membership in suspect classes). In effect, this allows one to run the formal tool non-exhaustively, observe a partial suspect set and then predict the remaining suspects without further calls to the engine, thus drastically reducing runtime.

Specifically, our contributions are as follows. First, we introduce a framework for learning design component dependencies based on suspect sets that have been resolved during past debugging sessions. The method represents these dependencies graphically, where each node in the representation corresponds to a design component previously seen in one or more suspect sets, and each edge between two components is assigned a conditional density quantifying the likelihood that one of the components will be in some set given the presence of the other. These conditional densities are learned by applying maximum a posteriori (MAP) estimates over the whole set of historical data. The graph essentially corresponds to an approximation of the true underlying suspect dependencies, where every suspect class is viewed as a clique. Second, we propose a method for predicting, given a partial suspect set, the suspects that are most likely to complement the remaining set. This is done via cost-effective single-pass belief propagation on the learned graph. Finally, we demonstrate that this process can reduce debugging and triage runtime by 15x on the average, while achieving a 91% average prediction accuracy, a remarkable feat. We note that although the proposed methodology is described using a formal debugger for simplicity, it is not confined to it — any bug hunting engine can be used as backbone.

The remainder of this paper is organized as follows. Section II contains preliminaries on automated design debugging. Section III presents the suspect dependency learning algorithm. Section IV shows how to predict solutions using the proposed representation. Section V gives experimental results and Section VI concludes the paper.

## II. Preliminaries

### A. SAT-based Design Debugging

A *failure* is said to occur in a design when the design's behaviour differs from the expected (golden) behaviour. A failure may manifest itself in multiple ways, such as an assertion failure due to a property violation, or a mismatch on the values of the primary output signals between the design under test (DUT) and the golden design. An *error trace* is associated with each failure which contains the sequence of vector values leading to the observed erroneous behaviour. The *bug* is the location in the RTL from which the failure originates due to incorrect logic/connections.

A SAT-based debugger can be used to find the set of all *suspects* corresponding to a failure. These are design locations

978-3-9819263-0-9/DATE18/©2018 EDAA

with which the erroneous behaviour could be corrected by applying a change at that location [2]. Because of the need to perform an exhaustive search involving multiple calls to a backend SAT-solver, runtimes and memory requirements of these tools can become prohibitive with increasing design size and longer error traces.

To alleviate this issue state-of-the-art debuggers incorporate techniques that eliminate the need to consider the entire error trace. In particular, *suffix window debugging* considers only a suffix of the error trace, implying that only suspects whose activation times are close to the failure time will be found [9]. This has the desirable effect of reducing runtime and memory use, but in general the resulting suspect set may be incomplete. If more suspects are needed then the suffix window can be iteratively expanded as described in [9], but at the cost of increasing runtime.

### B. Failure Binning and Triage

Regression testing for a design-in-progress will typically produce a large number of failures at a time, all of which need to be diagnosed and fixed. However, many of these failures will likely be caused by the same underlying bug, and so performing detailed debugging on all of them would compromise resources. *Failure binning* is the process of partitioning failures into groups such that two failures are in the same group if and only if they are likely to share a common root cause. Thus it is essentially a clustering process on the failures. It is the first stage of *failure triage*, which is the process of assigning failures to the engineer/team which is most likely to be able to diagnose and fix them [10], [11].

Failure binning involves running SAT-based debugging on each failure to identify all of its high-level suspect locations. This is currently the bottleneck of the failure binning process in terms of resource usage, and so any improvement here would result in significant improvement overall. For this reason we see failure binning as an ideal use case for a suspect prediction algorithm; it can achieve a considerable runtime reduction, and since failure binning is an approximate process to begin with, approximating the suspect sets does not have a major impact on the binning quality.

## III. Learning Suspect Relationships

While the technique of suffix window expansion generally reduces peak memory use, it may suffer from severe runtime inflation due to the need to iteratively expand the debug window with additional calls to the SAT solver in each iteration. Instead we propose that a single suffix window debug instance be run to obtain a partial suspect set, and the remaining suspects can be predicted probabilistically by learning from a history of debug sessions. The first step, which is described in more detail in the remainder of this section, is to estimate the probability of each pair of suspects occurring together based on the frequencies of the suspects in the debug history (analogous to [12]). In the next section this idea is extended to compute probabilities of suspects occurring in conjunction with an entire set of other suspects. Then the predicted suspects will be those which are most likely to occur with the subset obtained by suffix window debugging.

Formally, let $F_{hist} = \{F_1, ..., F_N\}$ denote a set of failures with corresponding suspect sets $S_{hist} = \{S_1, ..., S_N\}$, where each $S_i$ is a set of suspects $\{s_{i,1}, ..., s_{i,n_i}\}$. Moreover, let $SU = S_1 \cup S_2 \cup ... \cup S_N$. This constitutes the set of all distinct design locations that have ever been observed as suspects.

Suppose a new failure $F'$ is given. Its complete suspect set is $S'$, however, only a subset of these are observed, denoted $S'_{obs} \subseteq S'$. In practice $S'_{obs}$ could be obtained by debugging a suffix window of $F'$, in which case it would contain only the suspects whose activation times are close to the failure time. In general, however, no assumptions are made as to which suspects are in $S'_{obs}$ — the algorithm we present can be applied equally well to arbitrary subsets. The task we address next is to predict which suspects from $SU$ are in $S' \setminus S'_{obs}$.

A natural model for this purpose is a graph in which each node $i$ represents the suspect $s_i \in SU$ and each directed edge $(i, j)$ is weighted according to the strength of the implication of $s_i$ on $s_j$. Formally, we associate with every suspect $s_i \in SU$ a random variable $x_i$ which represents the event of $s_i$ being in $S'$. That is, $x_i = 1$ corresponds to $s_i \in S'$ and $x_i = 0$ corresponds to $s_i \notin S'$. Two directed edges are created between every pair of suspects (one in each direction), with the weight of edge $(i, j)$, denoted $w_{ij}$, equal to an estimation of the conditional probability $P(x_j|x_i)$. We use the notation $P(x_j|x_i)$ for convenience, which is understood to mean $P(x_j = 1|x_i = 1)$.

In order to compute these probabilities we consider the individual and pairwise frequencies of suspects in the training data. For individual suspects let $count(i)$ be the number of failures in the debug history that include $s_i$. Similarly, for every pair of suspects let $count(i, j)$ be the number of failures that include both $s_i$ and $s_j$. The most straightforward probability estimation would then be to assume that $P(x_i) = \frac{count(i)}{|S_{hist}|}$ and $P(x_i, x_j) = \frac{count(i,j)}{|S_{hist}|}$, giving

$$P(x_j|x_i) = \frac{count(i, j)}{count(i)} \tag{1}$$

Unfortunately this method can lead to severe overfitting to the training data when the $count$ values are small (which is frequently the case in practice). For instance, if suspect $s_1$ occurs only once in $S_{hist}$, say in suspect set $S_k$, and $S_k$ does not include $s_2$, then Eq. 1 would give $P(x_2|x_1) = 0$. However, it would be much too strong of a conclusion that $s_2$ cannot occur in the presence of $s_1$, having only ever observed $s_1$ once.

To deal with this issue we extend the preceding analysis with the standard technique of maximum a posteriori (MAP) inference [13]. The probabilities $P(x_j|x_i)$ can be viewed as underlying model parameters while the observed $count$ values are the data. The MAP technique selects parameter values which maximize the product of the data likelihood distribution and a prior distribution over the parameters. As for the former, when suspect $s_i$ is given then the event of observing $s_j$ is a Bernoulli random trial with probability $P(x_j|x_i)$. Thus $count(i, j)$ follows the binomial distribution

$$\begin{aligned} &P(count(i,j)|P(x_j|x_i), count(i)) = \\ &\mathcal{B}(x = count(i,j); n = count(i), p = P(x_j|x_i)) \end{aligned} \tag{2}$$

For the prior distribution, a Gaussian with a mean of 0.5 is used because no prior knowledge about the data is assumed apart from the fact that all values lie between 0 and 1. A variance of 0.2 was chosen as it is sufficiently large so as to allow the model to fit the data and sufficiently small so as to allow the model to generalize well. However, the precise value assigned to the variance was not found to have a major impact on the algorithm's overall performance.

In summary this gives

$$\begin{aligned} w_{ij} = P(x_j|x_i) = \underset{p}{\operatorname{argmax}}\, exp\left(-\frac{(p-0.5)^2}{0.4}\right) \\ \times \binom{count(i)}{count(i,j)} p^{count(i,j)}(1-p)^{count(i)-count(i,j)} \end{aligned} \tag{3}$$

Note that the initial proposition of Eq. 1 is equivalent to the maximum likelihood estimation (MLE) as it maximizes Eq. 2. MAP is much more appropriate as it avoids placing too much confidence on the parameter values for suspects which are rarely observed.

## IV. Suspect Prediction

The suspect graph can give the probability of observing a suspect conditioned on having observed any other suspect, but what we are actually interested in is the probability of observing a suspect conditioned on having observed the *set* of suspects $S'_{obs}$. In the first subsection we show how to compute these probabilities approximately using belief propagation on the suspect graph. This leads naturally to a ranking of all suspects according to these probabilities. The second subsection then describes a method to estimate the cardinality of $S'$, which is used to draw a cutoff line when deciding which suspects should actually be returned.

### A. Suspect Ranking

Consider a specific suspect $s_i \in SU \setminus S'_{obs}$. Letting $X'_{obs} = \{x_i : s_i \in S'_{obs}\}$, we are interested in the probability $P(x_i|X'_{obs})$. This is given by the following proposition [14].

**Proposition 1.** *Under the assumption of conditional independence between the graph edge probabilities, the probability of suspect $s_i$ being in $S'$ is*

$$P(x_i|X'_{obs}) = \begin{cases} 1, & s_i \in S'_{obs} \\ 1 - \prod_{s_j \in SU, j \neq i} (1 - P(x_i|x_j)P(j)), & s_i \notin S'_{obs} \end{cases} \quad (4)$$

*Proof:* For nodes in $S'_{obs}$ we are given $x_i$ so $P(x_i) = 1$. For nodes not in $S'_{obs}$, consider instead the complimentary probability $P(\bar{x}_i|X'_{obs})$. The only way the event $\bar{x}_i$ can occur is if $x_i$ is not observed with any of its neighbours. The probability $x_i$ being observed with neighbour $x_j$ is $P(x_i, x_j) = P(x_i|x_j)P(x_j)$, and if we assume that all neighbours act on $x_i$ independently then $P(\bar{x}_i|X'_{obs}) = \prod_{j \in SU, j \neq i}(1 - P(x_i|x_j)P(j))$. Eq. 4 then follows immediately. ∎

Because the suspect graph is a clique and not a DAG, many loops exist in the dependencies between suspects in Eq. 4. Therefore the additional approximation is made of only considering nodes being activated via edges $(i, j)$ where $i \in S'_{obs}$ and $j \in SU \setminus S'_{obs}$. The probabilities can then be computed with a single pass of belief propagation. For each unobserved suspect Eq. 4 simplifies to

$$P(x_i|X'_{obs}) = 1 - \prod_{s_j \in S'_{obs}} (1 - P(x_i|x_j)), s_i \notin S'_{obs} \quad (5)$$

Eq. 5 is computed for all $s_i \notin S'_{obs}$ and suspects are ranked in order of non-increasing $P(x_i|X'_{obs})$, which we will denote by $S_{rank} = (s_{r_1}, ..., s_{r_{|SU|}})$, where $1 \leq r_i \leq |SU|$. If a total of $k$ suspects are desired then the returned suspects are $(s_{r_1}, ..., s_{r_k})$. The overall process is illustrated in Example 1.

**Example 1.** *Consider the following history of suspect sets*

$$S_1 = \{s_1, s_4, s_5\}$$
$$S_2 = \{s_2, s_5\}$$
$$S_3 = \{s_1, s_2, s_3, s_4, s_5\}$$
$$S_4 = \{s_3, s_5, s_6\}$$
$$S_5 = \{s_1, s_3, s_4\}$$

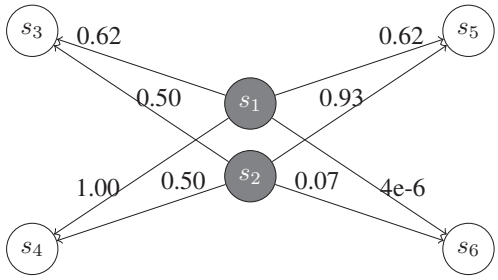


Fig. 1. Suspect graph for Example 1.

*Suppose the set we wish to predict and its observed subset are*

$$S' = \{s_1, s_2, s_4, s_5\}$$
$$S'_{obs} = \{s_1, s_2\}$$

*The first step is to compute the graph edge weights using Eq. 3. We consider only edges $(i, j)$ where $s_i \in S'_{obs}$ and $s_j \in SU \setminus S'_{obs}$. The relevant count values are shown in the following table.*

| $i$ | $count(i)$ | $count(1, i)$ | $count(2, i)$ |
|---|---|---|---|
| *1* | *3* | *3* | *1* |
| *2* | *2* | *1* | *2* |
| *3* | *3* | *2* | *1* |
| *4* | *3* | *3* | *1* |
| *5* | *4* | *2* | *2* |
| *6* | *1* | *0* | *0* |

*Consider the edge $(1, 3)$. Eq. 3 becomes*

$$w_{13} = \underset{p}{\text{argmax}}\, exp\left(-\frac{(p - 0.5)^2}{0.4}\right) \times \binom{3}{2} p^2(1 - p)^{3-2}$$
$$= 0.62$$

*The remaining weights are computed in a similar manner, and the resulting graph is shown in Fig. 1. Observed suspects are shown in gray while unobserved suspects are shown in white.*

*Next the suspect probabilities are computed using Eq. 5. For instance for $s_3$ we have*

$$P(x_3|x_1, x_2) = 1 - (1 - w_{13})(1 - w_{23})$$
$$= 0.809$$

*Doing this for the remaining suspects yields the probability scores*

$$P(x_4|x_1, x_2) = 0.999998$$
$$P(x_5|x_1, x_2) = 0.973$$
$$P(x_6|x_1, x_2) = 0.070$$

*from which we obtain the suspect ranking $S_{rank} = (s_1, s_2, s_4, s_5, s_3, s_6)$. If the engineer requests four suspects, $(s_1, s_2, s_4, s_5)$ would be returned.*

### B. Stopping Criterion

While belief propagation gives a ranking of the suspects by probability of being in $S'$, it says nothing as to how many of these suspects should actually be returned for $S'$, as the size of $S'$ is not known. We envisage this as a parameter that can be controlled by the engineer, as more suspects may only be required if the results achieved with partial suspect sets are unsatisfactory. Nonetheless, in many situations it would be useful to have an estimate of the actual number of suspects that would be returned by the debugger. To this end we propose

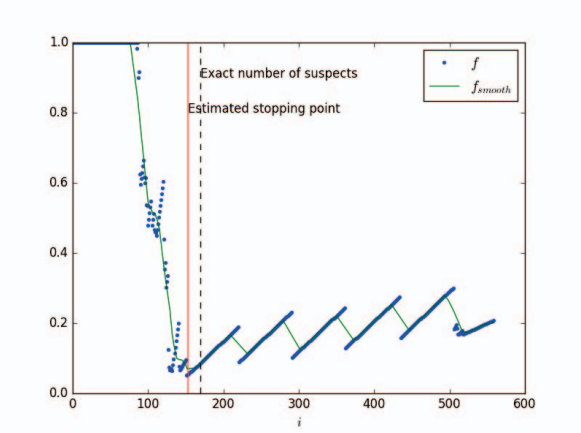


Fig. 2. Illustration of the stopping criterion. The function $f_{smooth}$ is plotted with the solid green line. The estimated stopping point is at the first local minimum (153) as shown by the solid red line, while the exact number of suspects is 169, shown by the dashed line.

the following criterion which dictates when to stop returning suspects.

**Definition 1.** *Stopping Criterion*

*After $S_{rank}$ is obtained by belief propagation as in Section IV-A, for each $i$ such that $|S'_{obs}| < i \leq |SU|$, define the function $f : \mathbb{Z} \rightarrow \mathbb{R}$ where*

$$f(i) = P(x_{r_i}|x_{r_1}, ..., x_{r_{i-1}}) = 1 - \prod_{1 \leq j < i} (1 - P(x_{r_i}|x_{r_j})) \quad (6)$$

*Because $f$ can be quite noisy we apply smoothing by taking the running average of $f(i)$. That is, let $f_{smooth}(i) = \frac{1}{2\delta+1} \sum_{j=i-\delta}^{i+\delta} f(j)$, where $\delta$ is a parameter whose value is determined empirically. The algorithm should then return the suspects $\{s_{r_1}, ..., s_{r_i}\}$ up to the smallest $i$ such that $f_{smooth}(i)$ is a local minimum.*

Eq. 6 is effectively the conditional probability of observing the $i^{th}$ ranked suspect having already observed the first $i-1$ ranked suspects. The justification for this criterion is that we stop returning more suspects at the earliest point that minimizes the probability of observing another suspect (which may not be a global minimum). This is illustrated in Fig. 2.

## V. Experimental Results

In this section we evaluate the performance of the proposed suspect prediction algorithm. Its prediction accuracy is measured on several different designs while key parameter values are varied. Runtimes of debugging suffix windows of various sizes are also compared against runtimes of full debugging. Next the error in the suspect set size estimation obtained from the stopping criterion is measured. Finally, failure binning is run using the predicted suspect sets, and the binnings obtained with the predicted suspects are compared against the binnings obtained with the ground truth suspect sets. All experiments are run on a i5-3570K 3.4 GHz machine with 16 GB of RAM.

### A. Data Collection Methodology

The proposed algorithm requires a history of failures and debug instances to learn from. For evaluation purposes we generated such a data set by injecting a variety of bugs into several *OpenCores* [15] designs as well as a design from an industrial partner. In order to avoid bias and to ensure the bug set included a diversity of design locations, bugs were injected randomly using a Python script. Each bug was created by selecting a module and signal assignment uniformly at random and forcing the signal assignment to 0 or 1. However, it is possible that such randomly generated stuck-at faults may not be representative of real bugs as a simple script cannot imitate the reasoning process of a design engineer. Therefore, several different types of bugs were created manually which more closely resemble human-introduced errors from our experience with industrial partners (including missing pipeline stages, incorrect operators in expressions, bad stimulus, complemented conditions in if-statements, incorrect state transitions, etc. [16]), and a combination of randomly- and manually-generated bugs are used for evaluation.

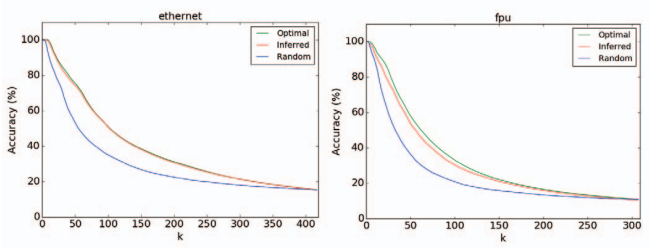


Fig. 3. Accuracy versus $k$ for the prediction, optimal, and random algorithms.

Each buggy design was then simulated and failures were debugged. The types of failures under consideration include `SystemVerilog` Assertion (SVA) failures and incorrect values on the primary output signals. The designs, their sizes, and total number of failures for each are shown in the first three columns of Table I.

### B. Suspect Prediction Accuracy

Let $k$ denote the number of desired suspects, and let $S'_{pred,k}$ denote the set of suspects returned by the prediction algorithm, so that $|S'_{pred,k}| = k$. Because $k$ is a parameter controlled by the engineer, the suspect prediction is evaluated over all values of $k$. We evaluate performance using the metric of *prediction accuracy*, defined as

$$accuracy(k) = \frac{|S'_{pred,k} \cap S'|}{k} \times 100\%$$

which is the percentage of returned suspects that are in $S'$.

We compare the prediction algorithm against optimal and random algorithms, which serve as upper and lower bounds, respectively, on reasonable prediction accuracies. The optimal algorithm is defined as the algorithm which, for a given $k$, always maximizes accuracy. That is, for $k \leq |S'|$ it returns only suspects in $S'$, and for $k > |S'|$ it returns all of $S'$ as well as $k - |S'|$ arbitrary suspects from $SU \setminus S'$. The random algorithm randomly chooses $k - |S'_{obs}|$ suspects from $SU \setminus S'_{obs}$.

To maximize the amount of test data we use a standard leave-one-out methodology whereby for each failure $F_i \in F_{hist}$ the model is trained on $F_{hist} \setminus F_i$ and tested on $F_i$. The resulting prediction accuracy is then averaged over all failures. In each case the sample $S'_{obs}$ contains 50% of the suspects in $S'$.

Fig. 3 plots the prediction accuracy against $k$ for $1 \leq k \leq |SU|$. Note that accuracy always begins at 100% for $k \leq |S'_{obs}|$ since $S'_{obs}$ is given, and it decreases as $k$ is increased. It can be seen that for all $k$ the prediction algorithm achieves an accuracy much closer to optimal than to random. This result is summarized in columns 5-7 of Table I which

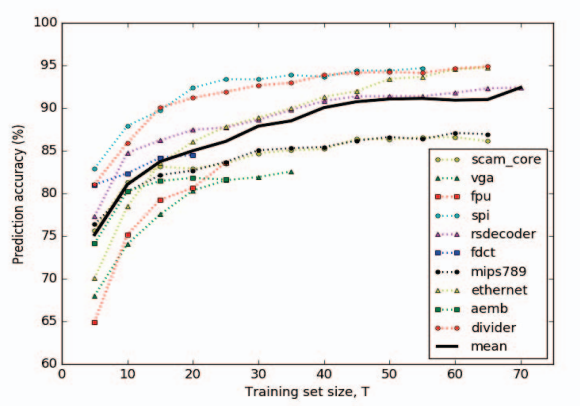


Fig. 4. Accuracy versus training set size with a sample size of 50%. Each design is shown in a dotted line while the mean is shown in the solid line.

give the mean accuracy over all $k$. On average the prediction algorithm achieves an accuracy of 44.9% compared to optimal and random values of 46.3% and 36.9%, respectively.

### C. Effect of Training Set Size

In this subsection we investigate how prediction accuracy varies with the number of debug instances that are used to learn the graph parameters, which we will denote by $T$. For each failure we test values of $T$ ranging from 5 up to a maximum of $|F_{hist}|-1$ in increments of 5. For a given $T$ and failure $F_i$ the training set was chosen randomly from $F_{hist} \setminus F_i$.

In order to isolate the effects of the suspect ranking from Section IV-A and the stopping criterion from Section IV-B, accuracy is measured at $k = |S'|$. The stopping criterion is evaluated independently in Section V-E. Fig. 4 plots accuracy against $T$, averaged over all failures for each design, and columns 8-10 of Table I give the numerical results. As would be expected accuracy increases rapidly with $T$ when $T$ is small, but only marginal improvement is achieved as $T$ gets larger. The figure suggests that a training set size of at least 20-30 debug instances would be desirable, but the algorithm can still work reasonably well with less training data.

### D. Effect of Sample Size

Next we measure the prediction accuracy for different sizes of the observed sample $S'_{obs}$, varying from 10% of $S'$ to 90% of $S'$ in 10% increments. Samples are chosen to include only the suspects whose activation times are closest to the failure time. This mimics the sample that would be obtained by suffix window debugging without the need to actually run it. As in Section V-C the accuracy is measured at $k = |S'|$. Fig. 5a plots the results for all sample sizes while columns 11-13 of Table I give the values for sample sizes of 40%, 60%, and 80%. The figure shows that accuracy increases monotonically with sample size as more suspects are given with $S'_{obs}$.

Fig. 5b gives an estimate of the runtime that would be saved by running only suffix window debugging rather than full debugging for various sample sizes. Due to the excessive amount of computation time that would be required, the suffix window debugging was not actually run for all window sizes and for all failures. Instead a single window size was chosen at random for each failure and debugged. The resulting suspect sample size is rounded to the nearest 10%, and the ratio of suffix debug runtime to full debug runtime is computed (the runtime of the suspect prediction being negligible in comparison). Then for each sample size in increments of 10%, the geometric mean of relative runtimes is plotted in Fig. 5b.

The figure indicates that a reasonable tradeoff between runtime and accuracy would be to use a sample size of around 50-70%. The mean accuracy in this range is 91% while the geometric mean runtime is 0.068 vs full debugging, corresponding to a 15x runtime reduction.

### E. Accuracy of Stopping Criterion

This section investigates how closely the stopping criterion presented in Section IV-B matches the actual size of $S'$. Let $k_{est}$ be the estimated stopping point. We measure the relative error in $k_{est}$ versus the ground truth value of $|S'|$ and take the average over all failures.

As noted in Section IV-B, a critical parameter in the stopping criterion is the range over which the incremental probability function is smoothed, $\delta$. With too little smoothing the algorithm could stop too early at a spurious local minimum caused by noisy data; with too much smoothing it might stop too late because the ideal local minimum was "smoothed away". By measuring the error for a range of $\delta$ values it was found that a value of about $\frac{|SU|}{50}$ works best for most designs.

Fig. 6 shows scatter plots of the actual and estimated set sizes for two designs. While it performs quite well for most failures, it tends to underestimate the size for larger suspect sets. This is a natural consequence of the fact that we stop at the *first* local minimum; in some cases it would be better to stop at a later local minimum, however, identifying the ideal local minimum remains a challenge. Column 14 of Table I gives the average error in $k_{est}$ for each design. Over all designs the suspect set size is predicted to within 24% of the actual value.

Next we examine how the error in $k_{est}$ affects the prediction of $S'$. Because the sizes of $S'_{pred,k_{est}}$ and $S'$ may now be different, a more appropriate metric than prediction accuracy is the Jaccard index between $S'_{pred,k_{est}}$ and $S'$. The results are given in column 15 of Table I. On average the predicted set matches the ground truth set with a Jaccard index of 0.72.

TABLE I. EXPERIMENTAL RESULTS ACROSS ALL DESIGNS AND PARAMETER VALUES

| Design | # gates | # failures | $\|SU\|$ | Mean accuracy over all $k$ (%) | | | Accuracy (%) at $k = \|S'\|$ | | | | | | $k_{est}$ error | Jaccard index at $k_{est}$ | Error in binning NMI | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Training set size | | | Sample size (%) | | | | | Sample size (%) | | |
| | | | | predict. | opt. | rand. | 10 | 20 | 30 | 40 | 60 | 80 | | | 40 | 60 | 80 |
| aemb | 20266 | 29 | 494 | 48.8 | 50.7 | 41.6 | 80.3 | 81.8 | n/a | 81.1 | 88.2 | 93.5 | .320 | .658 | .264 | .115 | .096 |
| divider | 10201 | 71 | 149 | 66.6 | 67.2 | 56.0 | 85.9 | 91.2 | 92.7 | 92.7 | 96.0 | 98.7 | .247 | .751 | .435 | .145 | .221 |
| ethernet | 45120 | 67 | 418 | 38.6 | 38.9 | 30.3 | 78.5 | 86.1 | 88.9 | 94.4 | 95.8 | 98.5 | .222 | .747 | .014 | .091 | .051 |
| fdct | 546765 | 22 | 559 | 34.7 | 38.2 | 29.2 | 82.4 | 84.5 | n/a | 83.5 | 89.0 | 95.6 | .207 | .727 | .224 | .156 | .072 |
| fpu | 82888 | 28 | 308 | 30.8 | 32.6 | 24.5 | 75.2 | 80.6 | n/a | 82.5 | 87.6 | 91.8 | .280 | .645 | .002 | .128 | .079 |
| mips789 | 55248 | 68 | 1063 | 44.3 | 45.7 | 35.7 | 80.2 | 82.7 | 85.1 | 85.4 | 88.8 | 94.5 | .176 | .731 | .043 | .087 | .029 |
| rsdecoder | 14842 | 72 | 1147 | 37.6 | 38.1 | 29.7 | 84.8 | 87.5 | 88.6 | 91.2 | 94.1 | 97.8 | .256 | .746 | .169 | .010 | .017 |
| scam_core | 1315446 | 68 | 485 | 52.1 | 53.6 | 42.7 | 81.4 | 82.9 | 84.7 | 84.2 | 89.3 | 97.6 | .255 | .717 | .351 | .051 | .106 |
| spi | 2528 | 60 | 228 | 59.9 | 60.5 | 50.6 | 87.9 | 92.4 | 93.4 | 94.6 | 95.1 | 97.5 | .241 | .735 | .011 | .059 | .059 |
| vga | 60533 | 38 | 890 | 35.8 | 37.5 | 28.8 | 74.1 | 80.4 | 81.9 | 80.0 | 85.9 | 94.6 | .177 | .702 | .431 | .067 | .126 |
| mean | | | | 44.9 | 46.3 | 36.9 | 81.1 | 85.0 | 87.9 | 87.0 | 91.0 | 96.0 | .238 | .716 | .195 | .091 | .086 |

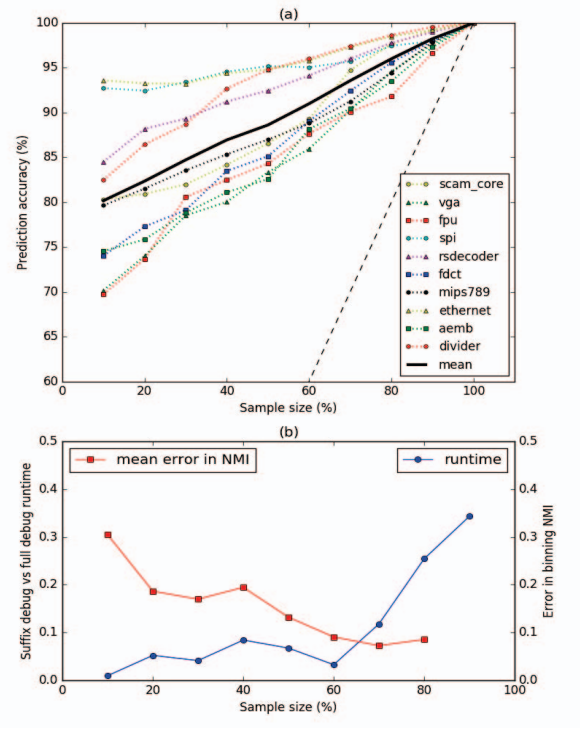


Fig. 5. Experimental results versus sample size as a percentage of $|S'|$. a) Prediction accuracy for all designs (dotted lines) and the mean (solid line). The minimum accuracy given from $S'_{obs}$ is shown by the lower dashed line. b) Runtimes vs full debug and mean error in failure binning NMI.

### F. Application to Failure Triage

One potential application for suspect prediction is failure triage, whereby failure binning would be performed using the predicted suspect sets rather than the suspect sets from full debugging. For experimental purposes we use the entire failure history $F_{hist}$ as a single triage instance, corresponding to the set of failures observed in regression testing. For each failure $F_i$ the suspect prediction is obtained in a leave-one-out manner by training on $F_{hist} \setminus F_i$. In this scenario it is not assumed that the engineer can provide a meaningful estimate on the number of suspects, so prediction is stopped at $k = k_{est}$. Bugs are considered only at the module level, that is, all failures originating from the same module are considered to be caused by the same bug and should be binned together.

Fig. 5b compares the binning using predicted suspect sets against the binning using exact suspect sets by plotting the relative error in the clustering normalized mutual information (NMI) between the two for a range of sample sizes, where binning is implemented as in [6]. Numerical data is given in columns 16-18 of Table I. We see again that there is a tradeoff between time spent debugging to obtain a larger sample and triage efficacy. At a sample size in the range 50-70%, failure binning with predicted suspect sets performs as well as binning with exact suspect sets to within 10%, while the mean debugging speedup is 15x.

## VI. Conclusion and Future Work

We present an algorithm to predict a failure's suspect set from a partial subset of suspects, which can be obtained by suffix window debugging. The algorithm first uses a history of debug sessions to learn the probability of observing each suspect given that another suspect has been observed. Then all potential suspects are ranked by probability of occurring

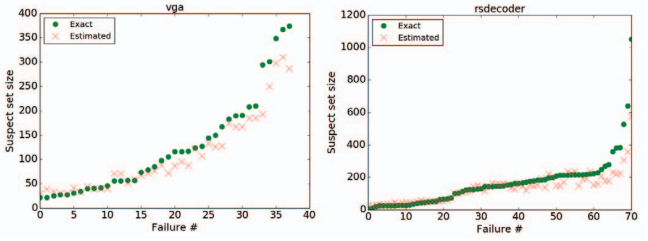


Fig. 6. Estimated and actual suspect set sizes for each failure in the vga and rsdecoder designs. Failures are ordered by non-decreasing exact set size.

with the given subset of suspects. Provided that a reasonable estimate of the number of suspects is given, this can achieve a prediction accuracy of up to 91% while reducing debugging runtime by 15x on average (Fig. 5). If the number of suspects is not known then an estimate is provided, which differs from the exact value by less than 24% on average. This is currently the bottleneck of the overall prediction accuracy, and so future work could be devoted to improving this estimate. Another avenue for future research would be to incorporate structural information such as dominance relationships so as to restrict the suspect graph and improve prediction accuracy.

## References


[1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[2] 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.

[3] 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.

[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 and A. Veneris, "Clustering-based failure triage for rtl regression debugging," in *Int'l Test Conference*, 2014, pp. 1–10.

[7] H. Mangassarian, L. Bao, and A. Veneris, "Debugging RTL using structural dominance," *IEEE Transactions on CAD*, vol. 33, no. 1, pp. 153–166, 2014.

[8] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Design Automation Conf.*, 2001, pp. 542–545.

[9] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *IEEE Transactions on CAD*, vol. 29, no. 11, pp. 1790–1803, 2010.

[10] Z. Poulos and A. Veneris, "Exemplar-based failure triage for regression design debugging," in *Journal of Electronic Testing*, 2016, pp. 125–136.

[11] H. Mangassarian, L. Bao, and A. Veneris, "Debugging RTL using structural dominance," *IEEE Transactions on CAD*, vol. 33, no. 1, pp. 153–166, 2014.

[12] 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.

[13] K. Murphy, *Machine Learning: a probabilistic perspective*. MIT Press, 2012.

[14] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 137–146.

[15] OpenCores.org, "http://www.opencores.org," 2006.

[16] Z. Poulos, R. Berryhill, J. Adler, and A. Veneris, "On simulation-based metrics that characterize the behavior of rtl errors," in *Proceedings of the Summer Computer Simulation Conference*, 2016, p. 14.