Suspect2vec: A Suspect Prediction Model for Directed RTL Debugging

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*—Automated debugging tools based on Boolean Satisfiability (SAT) have greatly alleviated the time and effort required to diagnose and rectify a failing design. Practical experience shows that long–running debugging instances can often be resolved faster using partial results that are available before the SAT solver completes its search. In such cases it is preferable for the tool to maximize the number of suspects it returns during the early stages of its deployment. To capitalize on this observation, this paper proposes a directed SAT–based debugging algorithm which prioritizes examining design locations that are more likely to be suspects. This prioritization is determined by `suspect2vec` — a model which learns from historical debug data to predict the suspect locations that will be found. Experiments show that this algorithm is expected to find 16% more suspects than the baseline algorithm if terminated prematurely, while still retaining the ability to find all suspects if executed to completion. Key to its performance and a contribution of this work is the accuracy of the suspect prediction model. This is because incorrect predictions introduce overhead in exploring parts of the search space where few or no solutions exist. `Suspect2vec` is experimentally demonstrated to outperform existing suspect prediction methods by an average accuracy of 5-20%.**

I. Introduction

Ensuring the functional correctness of modern electronic designs adds a layer of significant complexity to their development process, with verification and debugging often accounting for up to 70% of the design cycle [1]. The verification stage asserts that a design behaves as intended, while debugging focuses on localizing and correcting the root cause of verification failures.

Several automated debugging methods have been proposed to obviate the associated costs, with those based on formal methods such as Boolean Satisfiability (SAT) [2]–[5] proving to be particularly effective. Given a failing design and an error trace (*i.e.,* a sequence of input stimuli that can reproduce the erroneous behaviour), such tools identify a set of RTL locations where a modification can rectify the erroneous behaviour. These locations are referred to as *suspects*. In the SAT–based approach the debugging problem is encoded as a propositional formula with variables corresponding to potential suspect locations. The suspects are then identified by searching for satisfying assignments in which these variables are activated.

Recent work extends the SAT–based debugging methodology with techniques that allow it to scale to larger designs and longer error traces [6], [7]. However, further scalability can be achieved by exploiting the fact that these tools return solutions "on the fly" as they are discovered during the search process. This means that partial results are available before execution completes. If this partial suspect set is sufficiently large then it may enable the engineer to begin detailed debugging earlier. This could make further execution of the tool unnecessary and reduce the overall time needed to identify the bug. Therefore, the number of solutions that have been returned at each point in time becomes a key performance characteristic.

In the ideal case, SAT–based debugging would return most solutions at the very early phases of its search, with the remaining time spent only to prove that few or no other solutions exist. The extent of this behaviour can be quantified by the *average suspect recall*, where suspect recall is defined as the fraction of suspects (over the complete set) returned after a given amount of time, and the average is taken over the period of execution.

Prior work does not directly attempt to optimize debugging for this metric. To do so requires the SAT solver to adopt a search strategy that prioritizes regions of the search space where solutions are more abundant. Current debugging tools merely rely on the default heuristics of the solver [8] to define the search strategy and the order in which branching decisions are made. While these heuristics can be effective on a wide range of tasks, research in other domains [9] has shown that SAT solvers can be directed to a solution faster by taking problem-specific information into consideration when ordering the decision variables. We therefore expect that a similar approach may be a means to improve the average recall of the debugging algorithm.

The challenge lies in determining a good variable ordering for a specific debugging instance. To achieve this we develop a model which aims to *predict* the set of suspects the debugger will find given a subset of these suspects and a history of suspect sets from past debugging sessions. This suspect prediction model can then be integrated into the debugging algorithm as a mechanism to guide the underlying SAT search.

The suspect prediction task was first explored by [10] for the purpose of approximate debugging. The method mainly relies on heuristics to estimate the number of suspects in the approximation, which can lead to inaccurate predictions in certain cases. In contrast, this paper proposes a model where the prediction task is cast as a binary classification problem over all potential suspect locations, with the objective of separating the true suspects from the non-suspect locations. The classification is performed by learning representations of the suspects so as to maximize prediction accuracy on the historical data. This allows us to depart from heuristic–based approaches.

In summary, the contribution of this paper is two-fold. First, we present a model named `suspect2vec` which predicts the set of suspect bug locations given an initial sample of suspects. We empirically demonstrate the superiority of this approach over existing methods by an average prediction accuracy of at least 5%, with up to 20% gains seen at smaller sample sizes. Second, we describe a novel SAT–based debugging algorithm which imposes customized constraints on the branching order of suspect decision variables based on predictions from `suspect2vec`. Experiments show that this algorithm improves average suspect recall by 16%.

The remainder of this paper is organized as follows. Section II reviews the prerequisite topics of SAT–based de-

bugging, suspect set prediction, and embedding–based prediction models. Section III then describes the `suspect2vec` model while Section IV describes the new directed debugging algorithm. Section V provides an empirical evaluation of both contributions, and Section VI concludes the paper.

II. Preliminaries

A. SAT–based Automated Debugging

Design debugging is undertaken when a mismatch occurs between expected and observed signal values, often captured by assertions, golden-model checkers and/or scoreboards. Given the failing design's implementation, an error trace exposing a failure F, and a specification of the intended behaviour, a SAT–based debugger [3] returns the set of all suspects with respect to failure F, denoted by S. This is accomplished by introducing additional circuitry to model potential bugs at each design location, represented by Boolean variables $s_1, \ldots, s_n$. The augmented circuit and simulation vectors are then encoded in a Conjunctive Normal Form (CNF) formula Φ such that the location corresponding to s_i is a suspect if and only if there exists a satisfying assignment $\pi : M \rightarrow \{0, 1\}$ of Φ with $\pi(s_i) = 1$. Here M denotes the set of variables in Φ. Thus the suspect set S is obtained by repeatedly invoking a SAT solver on Φ until all satisfying assignments have been found.

B. Suspect Set Prediction

In this work we propose an improvement to the SAT–based debugging procedure by guiding the search so as to find more solutions in a given amount of time. To accomplish this we shift the focus to later stages of the development cycle when the design is closer to its final form and data from several historical debugging sessions are available. If this historical data is sufficiently diverse and representative with respect to bug types and buggy design locations then it can be used to predict the suspect set that a SAT–based debugger will find. These suspects should be prioritized accordingly when searching for satisfying assignments.

In detail, the suspect set prediction problem is defined as follows. Given a history of failures $\mathcal{F}_{hist} = \{F_1, \ldots, F_N\}$ along with their complete suspect sets $\mathcal{S}_{hist} = \{S_1, \ldots, S_N\}$ and a new failure F whose suspect set S is unknown, the formal debugger is run non-exhaustively only until a subset of the suspects, denoted $S_{obs} \subseteq S$, is found. This observed subset serves as an indicator which characterizes the failure and can be used to approximately predict the remainder of S without needing to run the formal tool any further. The motivation for this approach is that there tend to exist strong relationships between suspects such that if certain suspects occur in the solution set then other suspects are likely to occur as well. The historical data serves to identify these relationships.

This problem was recently addressed by [10] which proposes the following prediction mechanism. Let $SU = S_1 \cup S_2 \cup \ldots \cup S_N$ denote the set of all previously observed suspects in $\mathcal{S}_{hist}$. First, the suspect relationships are made explicit in the form of a graph containing all $s_i \in SU$ as nodes and directed edges between each pair of suspects (s_i, s_j) weighted by the conditional probability $P(s_j|s_i)$. These probabilities are estimated from $\mathcal{S}_{hist}$ using Bayesian inference. Next, all potential suspect locations are ranked by the probability $P(s_i|S_{obs})$, which is computed using a single pass of belief propagation on the suspect graph from the observed suspects to the unknown ones. Finally, a simple heuristic is proposed to estimate the cardinality of S and thereby draw a hard cutoff line to separate the predicted solutions from the non-solutions.

C. Embedding Representations and `Word2vec`

While the above methodology generally yields a good suspect ranking, the cardinality estimation often introduces significant error and limits the model's overall predictive ability. We instead propose an alternative model based on the concept of *suspect embeddings* — an idea inspired by the `word2vec` model [11] from Natural Language Processing (NLP), which we briefly review below.

`Word2vec` is a model for learning dense representations of words as vectors in a fixed-dimensional vector space, also called embeddings. Given a corpus of word tokens $w_1, \ldots, w_T$, vectors $\mathbf{v}_t$ and $\mathbf{v'}_t$ are associated with each word $w_t \in W$ in the corpus vocabulary W. These vectors play the role of model parameters and are learned so as to maximize the likelihood of the surrounding words $w_{t-c}, \ldots, w_{t+c}$ given the word w_t. Stated otherwise, the vector representation of w_t is trained to predict the words that occur in a context window of width $2c$ around w_t. It is this *predictive ability* that motivates the use of an embedding–based model for the suspect prediction task in the present work. More concretely, the `word2vec` model can be summarized by the following objective function:

$$\text{maximize } \frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log P(w_{t+j}|w_t) \tag{1}$$

where the conditional probability $P(w_{t+j}|w_t)$ is defined using the softmax function:

$$P(w_{t+j}|w_t) = \frac{\exp(\mathbf{v'_{t+j}} \cdot \mathbf{v_t})}{\sum_{i=1}^{|W|} \exp(\mathbf{v'_i} \cdot \mathbf{v_t})} \tag{2}$$

This formulation allows the model to be viewed as a single hidden layer neural network with the two vector representations for each word, $\mathbf{v}$ and $\mathbf{v'}$, as weights on the input and output side, respectively.

III. Suspect2vec

In this section we present a novel approach to the suspect set prediction problem which frames the problem in terms of binary classification. For each suspect $s_i \in SU$ we predict a binary label y_i which should be 1 if $s_i \in S$ or 0 otherwise, where S is the set we wish to predict. These labels must be predicted from a given subset S_{obs}, which is analogous to the prediction of a word given its context words. Moreover, the effectiveness of embedding representations in `word2vec` is grounded in the *distributional hypothesis* [12], which states that semantically similar words tend to occur in similar contexts. Likewise, related design suspects tend to occur in the same or similar suspect sets. This suggests that embeddings may also serve as a powerful means to capture suspect relationships for the suspect prediction task.

While parallels exist between the word prediction and suspect set prediction tasks, there are also crucial differences which necessitate considerable modifications to the presented model. Firstly, a suspect set is not perfectly analogous to a sentence of words because there is no meaningful order or locality defined between suspects, and the notion of a context window around a suspect is not applicable. The prediction model of `word2vec` (Eq. 2) is therefore not directly applicable, as it is only able to predict a single item given a single item. This is a consequence of defining the probability as a softmax function — it must be the case that $\sum_{w \in W} P(w|w_t) = 1$, meaning that at most one item in the vocabulary can have a strong (greater than 0.5) weight. This is at odds with our objective of predicting labels $y_i = 1$ for *all* $s_i \in S$. Thus, predicting a set of items given another set of items, both of arbitrary

size, requires a reformulation of the prediction model (Eq. 2) as well as the corresponding optimization objective (Eq. 1). The following two subsections address each of these aspects in detail.

A. Prediction Model

With each potential suspect location $s_i \in SU$ we associate an input and an output vector representation, denoted $\mathbf{v_i}$ and $\mathbf{v'_i}$, respectively. The conditional probability between suspects s_i and s_j is then defined as:

$$P(s_i|s_j) = \sigma(\mathbf{v'_i} \cdot \mathbf{v_j}) \tag{3}$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ denotes the logistic function. This is fundamentally different from Eq. 2 in that it treats all suspects independently, that is, the value of $P(s_i|s_j)$ does not impose any constraints on $P(s_k|s_j)$ for any $k \neq i$. This is a highly desirable property considering that we wish to predict sets of arbitrary size.

To deal with an arbitrarily sized set of *given* suspects, S_{obs}, we define a vector representation for S_{obs} as the arithmetic mean of suspect vectors in the set:

$$\mathbf{v_{obs}} = \frac{1}{|S_{obs}|} \sum_{s_i \in S_{obs}} \mathbf{v_i} \tag{4}$$

This is similar to the common practice in NLP of representing a sentence by the mean of word vectors within the sentence [13], and is motivated by the observation that sums of word vectors produce semantically meaningful vectors. We incorporate this idea directly into the prediction model by defining:

$$P(s_i|S_{obs}) = \sigma(\mathbf{v'}_i \cdot \mathbf{v}_{obs}) \tag{5}$$

where $\mathbf{v}_{obs}$ is given in Eq. 4. The final prediction can now be defined as $s_i \in S$ if and only if $y_i = P(s_i|S_{obs}) \geq 0.5$.

Like `word2vec`, `suspect2vec` can be expressed as a neural network with the embedding vectors aggregated into weight matrices $\mathbf{W}$ and $\mathbf{W'}$. Here $\mathbf{v}_i$ is the i^{th} column of $\mathbf{W}$, and $\mathbf{v'}_i$ is the i^{th} row of $\mathbf{W'}$. This is illustrated in Figure 1. The input is a binary vector $\mathbf{x}$ of length $|SU|$ representing the set S_{obs}, where $\mathbf{x}_i = 1$ if $s_i \in S_{obs}$ and 0 otherwise. $\mathbf{x}$ is first passed through a normalization layer so that $\mathbf{h_1} = \frac{1}{|S_{obs}|}\mathbf{x}$. The next layer multiplies this vector by weight matrix $\mathbf{W}$ to produce a hidden layer vector $\mathbf{h_2} = \mathbf{v}_{obs}$. The final layer multiplies $\mathbf{h_2}$ by the output weights $\mathbf{W'}$ and applies a logistic activation function, producing the output vector $\mathbf{y}$ with $\mathbf{y}_i = \sigma(\mathbf{v'}_i \cdot \mathbf{v}_{obs})$. Thus, the output is a vector of prediction scores $P(s_i|S_{obs})$ for all suspects $s_i \in SU$.

B. Training Procedure

This subsection describes the procedure used to learn the vectors $\mathbf{v}$ and $\mathbf{v'}$ such that Eq. 5 generates accurate predictions. This is achieved by optimizing an objective function that penalizes mispredictions on the historical (training) debug data $\mathcal{S}_{hist}$. For a suspect set $S_j \in \mathcal{S}_{hist}$, let $\hat{y}_i(S_j)$ denote the ground truth label for suspect s_i, where $\hat{y}_i(S_j) = 1$ if $s_i \in S_j$ or $\hat{y}_i(S_j) = 0$ otherwise. Because the prediction scores are defined by logits $y_i = P(s_i|S_{obs})$, we employ the standard *cross-entropy loss* to penalize the model for predicting discrepancies between y_i and $\hat{y}_i$. The cross-entropy loss is given by:

$$-\sum_{i=1}^{|SU|} \hat{y}_i \log y_i + (1-\hat{y}_i)\log(1-y_i)$$

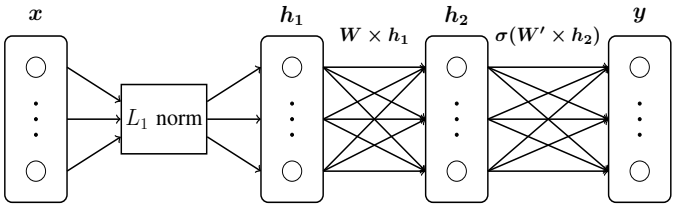

Fig. 1. The `suspect2vec` model as a neural network

Algorithm 1 SUSPECT2VEC-TRAIN($\mathcal{S}_{hist}, e, d, \alpha$)

```
for i ← 1 to |SU| do
    v_i ← RANDOM-VECTOR(d)
    v'_i ← RANDOM-VECTOR(d)
end for
for iter ← 1 to e do
    for each S_j ∈ S_hist do
        S_obs ← RANDOM-SUBSET(S_j)
        for i ← 1 to |SU| do
            v_i ← v_i − α∇_{v_i} L
            v'_i ← v'_i − α∇_{v'_i} L
        end for
    end for
end for
```

Ideally we would like to minimize this loss over all $S_j \in \mathcal{S}_{hist}$ and all possible observed subsets $S_{obs} \subseteq S_j$. This leads to the minimization objective:

$$L = -\sum_{S_j \in \mathcal{S}_{hist}} \frac{1}{2^{|S_j|}} \sum_{S_{obs} \subseteq S_j} \sum_{i=1}^{|SU|} \Big[\hat{y}_i(S_j)\log P(s_i|S_{obs}) + (1-\hat{y}_i(S_j))\log(1-P(s_i|S_{obs}))\Big] \tag{6}$$

The term $\frac{1}{2^{|S_j|}}$ is introduced to balance the total contributions of all S_j regardless of their varying sizes. Of course, Eq. 6 cannot be directly optimized or even computed in practice because the number of possible subsets S_{obs} is too large. However, it can be approximated by selecting many $S_{obs} \subseteq S_j$ at random. For each S_{obs}, the embedding vectors are updated according to gradient descent optimization, where the gradients of Eq. 6 are as follows:

$$\nabla_{\mathbf{v}_i} L = \begin{cases} \frac{1}{|S_{obs}|}\mathbf{v'}_i\left[\sigma(\mathbf{v'}_i \cdot \mathbf{v}_{obs}) - \hat{y}_i\right], & s_i \in S_{obs} \\ 0, & s_i \notin S_{obs} \end{cases}$$
$$\nabla_{\mathbf{v'}_i} L = \mathbf{v}_{obs}\left[\sigma(\mathbf{v'}_i \cdot \mathbf{v}_{obs}) - \hat{y}_i\right] \tag{7}$$

Algorithm 1 summarizes the training procedure. The subroutine RANDOM-VECTOR(d) returns a random vector of length d, while the subroutine RANDOM-SUBSET(S) generates a random subset of S by independently including or excluding each $s_i \in S$ with probability 0.5. The number of iterations e, the learning rate α, and the dimensionality of the vectors d are all hyperparameters which can be tuned to the specific data set.

IV. Directed Debugging using Suspect Prediction

In this section we describe a new SAT–based debugging procedure with improved average suspect recall. In effect, by optimizing for this metric we increase the expected number of returned suspects should the tool be terminated early or should detailed debugging commence before the tool completes. The proposed algorithm leverages a suspect prediction model to estimate which areas of the search space are most likely to contain solutions and should therefore be explored first. It then dictates this search strategy to the SAT solver by

Algorithm 2 DIRECTED-DEBUG(Φ, m)

```
1: S_obs ← ∅
2: A ← ∅
3: while |S_obs| < m do
4:     s ← SOLVE(Φ, A)
5:     if s ≠ ⊥ then
6:         S_obs ← S_obs ∪ {s}
7:         Φ ← Φ ∧ ¬s
8:     else
9:         return S_obs
10:    end if
11: end while
12: S_pred ← SUSPECT-PREDICTION(S_obs)
13: A ← {¬s : s ∉ S_pred}
14: while True do
15:    s ← SOLVE(Φ, A)
16:    if s ≠ ⊥ then
17:        S_obs ← S_obs ∪ {s}
18:        Φ ← Φ ∧ ¬s
19:    else if A ≠ ∅ then
20:        Φ ← Φ ∧ ⋀_{s∈S_pred} ¬s
21:        S'_pred ← SUSPECT-PREDICTION(S_obs)
22:        if S'_pred ≠ S_pred then
23:            A ← {¬s : s ∉ S'_pred}
24:            S_pred ← S'_pred
25:        else
26:            A ← ∅
27:        end if
28:    else
29:        return S_obs
30:    end if
31: end while
```

imposing constraints on the order in which suspect variables are branched on as true. We expect this decision ordering to find most solutions earlier than the default order which is governed by the general-purpose VSIDS heuristic [8], as the latter has no understanding of the underlying problem or the significance of the variables it is ordering.

Enforcing the desired decisions is implemented by adding assumption constraints to the SAT instance, which are single-literal clauses that can be added or removed between queries. The justification for this approach is two-fold: it is a non-intrusive scheme in that it allows the solver to be treated as a black box, and it also allows constraints to be conveniently removed in later SAT queries. This latter point is critical because we still wish for all solutions to be found; therefore, any constraints that are imposed to guide the search must ultimately be removed.

One might naturally hope to achieve the desired behaviour by ordering the suspect variables by probability scores (see Eq. 5), $s_{p_1}, \ldots, s_{p_{|SU|}}$, and successively calling the SAT solver with the assumption literal s_{p_i} for each $i = 1, \ldots, |SU|$ in this order. Unfortunately, we found experimentally that this method can severely slow down the overall search procedure because the frequent removal of constraints can be difficult for an incremental SAT solver to deal with. Instead, we take an approach that only infrequently requires removing constraints. Rather that forcing the solver to immediately branch true on the next predicted suspect, we prevent the solver from branching true on any of the non-predicted suspects by imposing the set of blocking assumptions $\{\neg s_i : P(s_i|S_{obs}) < 0.5\}$.

This is described in detail in Algorithm 2 and works as follows. The input consists of Φ, the CNF encoding of the augmented circuit as described in [3], and the parameter m which controls the size of the initial solution subset. This subset is found in lines 3-11 using the original debugging algorithm from [3]. The subroutine SOLVE($\Phi, \mathcal{A}$) returns a suspect variable corresponding to a satisfying assignment of Φ under assumption set $\mathcal{A}$, or the null value $\perp$ if none exists. Each time a solution s is found it is added to the solution set S_{obs} and blocked with the hard clause $\neg s$ to prevent it from recurring in subsequent calls. Then in lines 12-13 the prediction is run on S_{obs}, and all $s \notin S_{pred}$ are temporarily blocked by assumptions. Lines 14-18 find all solutions until the solver returns UNSAT. At this point (lines 19-27) the prediction is rerun because new solutions have been found, so S_{pred} is updated using this additional information. Line 23 removes the blocking assumptions of any newly predicted suspects, and the process is repeated. Eventually the prediction stabilizes and lines 25-26 are executed. Here all assumptions are removed and the solver searches for any remaining solutions that were previously blocked. Once all such solutions have been found lines 28-29 are executed and the search terminates.

Note also that the SUSPECT-PREDICTION subroutine does not incur significant runtime overhead because the training step (Algorithm 1) can be performed offline. DIRECTED-DEBUG only needs to run the prediction step which is extremely efficient.

We now argue the soundness and completeness of DIRECTED-DEBUG. In this context soundness means that all returned suspects are possible bug locations; completeness means that all possible bug locations are returned. Let S_{base} and S_{new} denote the suspect sets returned by the baseline algorithm of [3] and DIRECTED-DEBUG, respectively. For notational convenience we also define $\text{CNF}(\mathcal{A}) = \bigwedge_{l \in \mathcal{A}} l$.

Lemma 1. $S_{base} \supseteq S_{new}$

Proof: For each $s_i \in S_{new}$ there exists a satisfying assignment π with $\pi(s_i) = 1$. Since $\pi \models \Phi \wedge CNF(\mathcal{A})$ then $\pi \models \Phi$ so $s_i \in S_{base}$. ■

Lemma 2. $S_{base} \subseteq S_{new}$

Proof: This can be seen from line 26 where $\mathcal{A} = \varnothing$ and the solver is then run on $\Phi \wedge \bigwedge_{i=1}^{k} \neg s_{b_i}$, where $s_{b_1}, \ldots, s_{b_k}$ are the suspects that have previously been blocked. Thus any $s_i \in S_{base}$ must either be found after line 26 executes or have been blocked on line 7, 18, or 20. In each of these cases it is clear from the pseudocode that $s_i \in S_{new}$, except for suspects blocked on line 20. However, in this last case we know that there does not exist $\pi \models \Phi$ with $\pi(s_i) = 1$ because $\Phi \wedge CNF(\mathcal{A})$ is UNSAT, and $\neg s_i \notin \mathcal{A}$ since $s_i \in S_{pred}$, implying that $s_i \notin S_{base}$. Therefore $s_i \in S_{new}$ for all $s_i \in S_{base}$. ■

Theorem 1. DIRECTED-DEBUG *is both sound and complete.*

Proof: Lemmas 1 and 2 imply that $S_{base} = S_{new}$, and [3] is sound and complete. The theorem follows immediately. ■

V. EXPERIMENTAL RESULTS

In this section we evaluate the performance of `suspect2vec` and DIRECTED-DEBUG on the same data set as used in [10], which consists of 10 different designs from OpenCores [14] and industry. Each design comprises multiple bugs injected by randomly forcing signals to 0 or 1, as well as a variety of manually generated bugs which more closely resemble human–introduced errors. For each bug, all unique testbench errors were initially debugged using a SAT–based debugger implemented as described in [3] to extract the ground truth suspect sets. All experiments are run on a i5-3570K 3.4 GHz machine with 16 GB of RAM.

A. Suspect2vec Prediction

We begin with an evaluation of the `suspect2vec` prediction model in comparison to the baseline of [10]. We employ a leave-one-out evaluation methodology whereby $|\mathcal{S}_{hist}| - 1$ failures are used as training data, and the remaining failure serves as a test point. Note that this is equivalent to k-fold cross validation with $k = |\mathcal{S}_{hist}|$. The prediction task is set up by selecting a sample S_{obs} containing the first 50% of the suspects that are found by the SAT–based debugger. This choice of S_{obs} replicates the sample that would be obtained in DIRECTED-DEBUG — our intended application for suspect prediction. Hyperparameter settings include a learning rate α of 0.01, dimensionality d of 20, and $e = 4000$ iterations. While this dimensionality may seem extremely small compared to typical values used with other neural embedding models such as `word2vec`, no empirical improvement was observed at higher values. The representational capacity of $d = 20$ appears to be sufficient for the purposes of suspect prediction where the number of objects $|SU|$ is typically not much greater than 1000.

Performance is measured using three metrics: precision (the fraction of predicted suspects that are true solutions), recall (the fraction of true solutions that are predicted), and F_1 score (the harmonic mean of precision and recall). The F_1 score provides an overall measure of classification accuracy which balances the values of precision and recall. On each design we report the mean of these metrics over all failures. The results are given in Table I, with the mean precisions, recalls, and F_1 scores in columns 5-10. The results clearly indicate that `suspect2vec` outperforms the baseline, with a greater average F_1 score on nearly every benchmark. We also find it informative to measure the relative error in the cardinality of the predicted set, which is given in columns 11-12. However, due to a large number of outliers in this metric (for both prediction methods) we report the median over all failures. It can be seen that `suspect2vec` generally predicts the suspect set cardinality more accurately than the baseline, which largely explains its greater F_1 score.

We also consider how each model performs with varying amounts of training data. This is controlled by reducing the number of folds, k, as the fraction of data that is used for training is $\frac{k-1}{k}$. Columns 13-16 give the prediction F_1 scores at $k = 2$ and $k = 5$. We observe that `suspect2vec` still outperforms the baseline by 4% even when the training set size is halved at $k = 2$.

Finally, we evaluate the predictions with different sizes of the observed sample, S_{obs}. Figure 2 plots the mean F_1 scores over all designs against the sample size as a percentage of the complete suspect set size from 10% to 90% in increments of 10%. The figure indicates that the performance gap widens at smaller sample sizes, reaching as much as 20% at a 10% sample size, while remaining near 5% at larger samples.

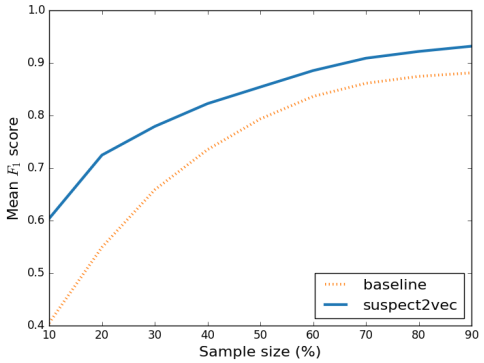

Fig. 2. Mean prediction F_1 scores versus $|S_{obs}|$.

B. Directed Debugging Evaluation

In this subsection we evaluate the performance of DIRECTED-DEBUG against the baseline SAT–based debugging algorithm of [3]. Unlike prior work, of primary interest is the characteristics of the algorithms' suspect recall versus time curves rather than their total runtimes. Intuitively, an algorithm that returns more suspects near the beginning of the search would be preferable, even if the total runtime is not improved, as it may allow detailed debugging to begin earlier. This notion can be formalized by measuring the area under the recall-time curve, which we denote by the symbol R. Each time a suspect $s_i \in S$ is returned we define a coordinate (i, t_i), where t_i is the time at which the i^{th} suspect is returned and $i = 1, \ldots, |S|$. Then the performance metric is computed as

$$R = \sum_{i=1}^{|S|-1} \frac{i}{|S|} \frac{t_{i+1} - t_i}{T} + \frac{T - t_n}{T} \tag{8}$$

where T is the total runtime of the baseline algorithm. Note that due to normalization of the time coordinates, R can also be interpreted as the average or expected recall that would be obtained if the debugging algorithm were terminated early. This is illustrated in Figure 3, where we plot the recall-time curves of both DIRECTED-DEBUG and [3] on an exemplary failure from the ethernet design.

In our experiments the SUSPECT-PREDICTION subroutine of DIRECTED-DEBUG is implemented using `suspect2vec`, as it generally offers better performance over alternative methods. MiniSat [15] is used as the backend SAT solver. The remaining configuration choice is the value of the hyperparameter m. If set too high then the solver will have already explored most or all of the search space before the prediction stage kicks in, rendering it ineffective. On the other hand, if set too low then the prediction may exhibit poor accuracy, causing many solutions to be incorrectly blocked and impairing the algorithm's performance. To balance these two effects we heuristically set m to half the average suspect set size in $\mathcal{S}_{hist}$.

TABLE I. COMPARISON OF `SUSPECT2VEC` (S2V) AND THE BASELINE OF [10] (BASE) AT 50% SAMPLE SIZE

Design	# gates	# failures	$\|SU\|$	Precision		Recall		F_1 score		Size error (median)		2 folds		5 folds	
				s2v	base	s2v	base	s2v	base	s2v	base	s2v	base	s2v	base
aemb	20603	29	592	0.768	0.810	0.784	0.687	0.749	0.727	0.162	0.264	0.708	0.696	0.732	0.728
divider	10334	71	153	0.944	0.965	0.917	0.688	0.923	0.794	0.074	0.318	0.846	0.729	0.894	0.774
ethernet	82803	66	533	0.935	0.863	0.929	0.837	0.926	0.831	0.052	0.237	0.762	0.717	0.873	0.794
fdct	546878	27	617	0.921	0.911	0.877	0.860	0.885	0.878	0.168	0.136	0.867	0.851	0.883	0.865
fpu	82938	27	367	0.786	0.752	0.775	0.734	0.753	0.726	0.227	0.245	0.626	0.662	0.738	0.707
mips789	55248	68	1143	0.889	0.905	0.791	0.722	0.826	0.797	0.152	0.241	0.806	0.787	0.821	0.794
rsdecoder	14890	72	1415	0.893	0.855	0.892	0.818	0.871	0.816	0.120	0.171	0.845	0.789	0.863	0.806
scam_core	1315446	66	444	0.909	0.947	0.933	0.748	0.910	0.827	0.058	0.223	0.825	0.739	0.847	0.800
spi	2536	60	242	0.963	0.934	0.922	0.682	0.938	0.773	0.048	0.297	0.799	0.690	0.819	0.721
vga	44579	38	933	0.728	0.787	0.852	0.769	0.761	0.766	0.309	0.212	0.682	0.731	0.710	0.737
mean				0.874	0.873	0.867	0.754	0.854	0.793	0.137	0.235	0.777	0.739	0.818	0.773

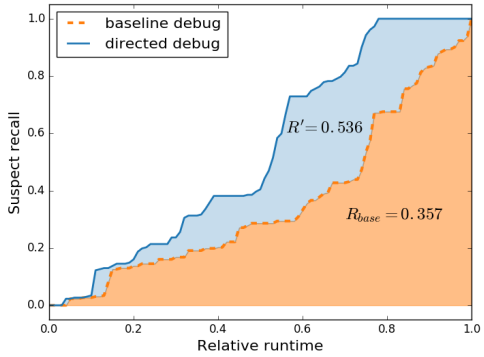

Fig. 3. Example of recall versus time curves and the average recall R for a failure from the ethernet design.

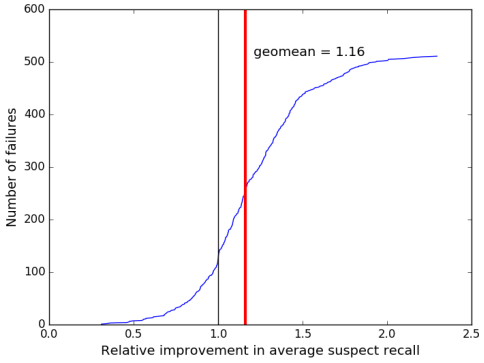

Fig. 4. Improvements in average suspect recall for all benchmark instances.

For each failure $F_j \in \mathcal{F}_{hist}$ we compute the metrics R^j_{base} and R^j_{new} from Eq. 8 for [3] and DIRECTED-DEBUG, respectively. We then compute the relative improvement which is given by the ratio R^j_{new}/R^j_{base}. Each debugging instance is run with a maximum time limit of 1 hour. Note that Eq. 8 is well–defined even when the run is terminated at the time limit, so such instances are included in the reported results. The results are depicted in Figure 4, which plots the relative improvement on the x-axis and the cumulative number of instances (aggregated over all designs) with an improvement of at least this value on the y-axis.

Table II presents numerical results for the designs individually, where column 3 summarizes average recall by the geometric mean over all of failures. Columns 4-6 describe the total runtimes of the algorithms, with columns 4 and 5 giving the number of debug instances that were completed within the time limit. Of such instances, the geometric mean of relative runtime is given in column 6. These results indicate similar total runtimes between the two algorithms.

The average recall results exhibit a greater variance,

TABLE II. IMPROVEMENTS IN AVERAGE RECALL AND RUNTIME FOR ALL BENCHMARKS

Design	m	Geomean of R^j_{new}/R^j_{base}	# failures completed		Relative runtime
			base	new	
aemb	71	1.32	28	27	0.79
divider	28	1.35	71	71	0.80
ethernet	39	1.26	66	66	0.79
fdct	42	1.16	27	27	0.93
fpu	18	1.24	27	27	0.79
mips789	108	0.95	50	50	0.95
rsdecoder	99	1.03	53	49	1.01
scam_core	61	1.09	58	58	0.95
spi	40	1.38	60	60	0.76
vga	71	0.95	29	30	0.99

but 76% of test instances see positive improvements under DIRECTED-DEBUG, with an overall geometric mean improvement of 16%. Closer inspection of the cases that are negatively impacted reveals that many are due to inaccurate suspect predictions causing solutions to be incorrectly blocked. A larger value of the parameter m could potentially mitigate this issue, but a fully automated method to select the optimal value on a failure-by-failure basis remains an open problem.

VI. CONCLUSIONS

In this paper we argue that a debugging algorithm that returns more suspects earlier in its search can aid an engineer in localizing the bug faster using partial results. This addresses the gap in the existing literature which focuses primarily on reducing the runtime of a complete search. We present an optimized debugging algorithm that searches for solutions with guidance from a suspect prediction model, which predicts the set of solutions that will be found given a subset of these solutions and historical debugging data. We further show how the suspect prediction problem can be solved using an embedding–based binary classification model named `suspect2vec`. Our experiments demonstrate that `suspect2vec` vastly outperforms existing methods by an average prediction F_1 score of 5-20%, while the directed debugging algorithm can improve the average suspect recall by 16%.

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] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita, "A formal approach for debugging arithmetic circuits," in *tcad*, 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," *tcad*, 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 *itc*, 2008, pp. 1–8.

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

[6] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *tcad*, vol. 29, no. 11, pp. 1790–1803, 2010.

[7] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1597–1608, 2009.

[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.

[9] O. Shacham and E. Zarpas, "Tuning the vsids decision heuristic for bounded model checking," in *Microprocessor Test and Verification: Common Challenges and Solutions, 2003. Proceedings. 4th International Workshop on*. IEEE, 2003, pp. 75–79.

[10] N. Veira, Z. Poulos, and A. Veneris, "Suspect set prediction in rtl bug hunting," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 1544–1549.

[11] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[12] M. Sahlgren, "The distributional hypothesis," *Italian Journal of Disability Studies*, vol. 20, pp. 33–53, 2008.

[13] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning*, 2014, pp. 1188–1196.

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

[15] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.