An Extensible Perceptron Framework for Revision RTL Debug Automation

John Adler[1], Ryan Berryhill[1], Andreas Veneris[1,2]

***Abstract*—Automated debugging techniques can significantly reduce the manual effort required to localize RTL errors. These techniques return to the user a set of RTL locations where a change can correct erroneous behavior. However, each location must be manually investigated. This problem is exacerbated by the increasing amount of failures in the modern regression verification cycle. Recent work in clustering-based revision debugging mitigates this cost by ranking revisions based on their likelihood of having introduced an error. This work presents a perceptron-based approach to revision debugging that can be extended to leverage the revision history of a design directly. Perceptrons are trained using labeled revisions from the design history. They are then used to predict the probability that a revision has introduced an error. The proposed methodology performs competitively with the state-of-the-art, but can be extended to handle more features. This allows for an automated regression debug flow integrated with Version Control and Issue Tracking Systems.**

I. Introduction

The modern hardware design cycle is bottlenecked by functional verification, which accounts for up to 70% of the total effort [1]. The majority of this cost is spent on debugging: localizing and correcting design errors [2]. In order to alleviate the engineering effort spent of these tasks, automated techniques have been developed.

Verification can be performed either *on-line* or *off-line*. On-line verification involves the engineer analyzing the design functionality through means such as simulation and model checking. When a failure is observed, an error-trace is returned that exposes the erroneous behavior. This is followed by fine-grain debugging, where the engineer seeks to find the root cause of the error. Automated debugging tools based on Boolean Satisfiability (SAT) solving [3] can be used to accelerate the process. Using the error-trace(s), the tool finds all locations in the design Register-Transfer Level (RTL) where a change can be made to correct the behavior. For the convenience of the engineer, these are typically mapped to lines in the hardware description language (HDL) source code for the design.

Conversely, off-line (or regression) verification runs extensive test suites that exercise a large portion of the design functionality. When the process completes, engineers perform coarse-grain debugging by analyzing and parsing the simulation logs and error messages. Automation is critical to the process due to the large volume of data. However, relatively little automation is available today, often consisting of simple rule-based approaches that assign each failure to an engineer who must perform fine-grain debugging.

To accelerate off-line verification, a recent development in revision debugging [4] introduces a clustering-based algorithm that leverages information from a Version Control System (VCS). Given a list of revisions, the tool ranks each revision in the VCS according to its expected likelihood of being responsible for the observed failure. An extension to this work [5] makes use of the non-linear nature of typical modern VCSs, such as Git [6]. By adding branching information to the algorithm, branches are ranked in addition to revisions. Both branch and revision rankings allow the engineer to pinpoint the root cause of the failure with less effort, thus reducing the cost associated with debugging the design.

These techniques, while providing valuable information, are inflexible, and do not make full use of modern development practices. An Issue Tracking System (ITS) is often used in conjunction with a VCS in order to enrich the information provided by the latter [7]. Most prominently, revisions and branches can be labeled, most often with whether they correspond to a bugfix or a feature addition. Additionally, [4], [5] provide no means of incorporating the wealth of information available in a VCS, such as commit timestamps, commit authors, commit logs, etc. Another issue with these techniques is that as the number of revisions increases, performance decreases, as revisions are ranked on a relative rather than absolute scale.

To address these shortcomings, this work presents an extensible framework based on a perceptron [8], rather than the clustering-based approach of [4], [5]. A perceptron is trained using historical data on design errors and their eventual fixes, available from the VCS and ITS for each design. The trained perceptron can then used to predict the probability that future revisions have introduced a bug. Ranking of revisions can be accomplished by sorting revisions on this probability. This model has the capability of being extended to learn from the rich additional data available in both the VCS and ITS, unlike the fixed clustering-based approach.

Perceptrons based on logistic regression are trained per design using a mixture of historical design data and results from an automated debugging tool. Each training sample revision is tagged with matching suspects, allowing the perceptron to learn a relationship between revisions and suspects. Classification performance enhancements, including using an SVM and a more involved matching metric, are also presented. This work is applicable to both on-line and off-line verification, as it can speed up error localization, but is more conducive to the coarse-grain nature of off-line verification by providing broad guidance on which revisions to prioritize.

Experimental results demonstrate that this methodology can perform competitively to the state-of-the-art clustering revision debug methodology. On average, ranking is only 2.83 times worse than the clustering-based approach, with a 75% decrease in average runtime. However, the benefits of the perceptron-based approach can be seen in cases where a wealth of training data is available, resulting in up to 33% better ranking performance. In addition, the perceptron-based approach can be improved with additional training samples and the addition of new features to the model, vectors of improvement not available to the clustering-based approach.

The remainder of this paper is organized as follows. Section II presents relevant background information. Section III presents the novel methodology of perceptron-based revision debug. Section IV extends the framework to improve classification performance. Section V summarizes experiments results. Finally, Section VI concludes the work.

[1]University of Toronto, ECE Department, Toronto, ON M5S 3G4 ({adler, ryan, veneris}@eecg.toronto.edu)

[2]University of Toronto, CS Department, Toronto, ON M5S 3G4

II. Preliminaries

A. SAT-Based Debugging

SAT-based debugging is central to both the work presented here and that of [4], [5]. Consider a circuit with multiple errors in the RTL. When off-line verification detects a failure by means such as an observation value mismatch or a firing assertion, an error trace is returned that exposes the failure. Let $F = \{f_1, ..., f_{|F|}\}$ represent the failures returned by an off-line verification run. Note that different failures may be caused by different errors in the RTL.

Given an error trace exposing a failure, SAT-based debugging tools [3] identify candidate lines where a fix can be implemented to correct the failure. For a failure f_i, the tool returns a set of candidate lines $S_i = \{s_1^i, ..., s_{|S_i|}^i\}$ in the HDL representation of the circuit. The set S_i is also referred to as the *suspect set*, and accordingly each element of S_i is called a *suspect*. A suspect could be a module definition, module instantiation, expression, etc., but ultimately each is mapped to a range of lines in the design's HDL. As SAT-based debugging techniques are exhaustive, the suspect set contains every such line, and therefore is guaranteed to include the actual error source. In practice, it tends to contain many other lines where a change can be made to merely mask the failure for the particular error trace but cannot correct it for all other traces. As an engineer must manually investigate the suspects in order to correct the failures, it is desirable to provide an accurate means of filtering out suspects that are false positives using a ranking system.

B. Version Control Systems and Issue Tacking Systems

Version control systems are part of the modern software configuration management workflow. Successive changes to design files, known as *revisions* or *commits*, are tracked. This revision history allows different versions of the code to be examined and compared. Traditionally, each revision has associated with it a unique identifier, a commit log (*i.e.,* a user-specified description of the changes), and a list of changes. Changes are usually represented using a line-based `diff`.

Modern VCSs support a *branching* scheme, which allows multiple developers to work on a design in tandem (fixing bugs, adding features, etc). Following proper coding practices, each branch should isolate development on a single feature or bugfix. Using this scheme, revisions are pushed to a branch, rather than the mainline, *i.e.,* the master branch. Once development on a branch is complete, it is merged onto the mainline. Branches still under development, or ones not yet merged onto the mainline are termed *redundant* while revisions within those branches are termed redundant revisions.

Issue tracking systems are used to complement limitations of VCSs by enriching the information available, through the use of *issues*. One or more branches can be associated with each issue, with the finished branch resolving the issue. Relevant to this work, issues—and, by association, branches—can be tagged with various descriptors, with "bugfix" or "feature" being the two most prominent. ITSs can also allow for relationships between issues to be described: one branch might fix a bug introduced in a previous branch, for example.

C. Clustering-Based Revision Debug

Previous work in revision debug [4], [5] introduces a clustering-based methodology for ranking revisions and branches, allowing the engineer to rank revisions more likely to have introduced an error. This process is broadly separated into three steps: suspect clustering, classification, and weighted ranking.

Suspect clustering is performed independently of the VCS, relying only on results from an automated debugging tool [3]. The tool is run on the failures from regression, returning a set of suspects for each failure. Affinity propagation clustering [9] is used on these suspects to automatically identify groups of similar suspects, as seen in Figure 1. To this end, the lines for each suspect S_i are plotted onto a space, with one file per axis. In this example, five suspects are plotted across two source files. Exemplars (cluster centers) will be located near areas of high overlap, labeled as C in the left cluster and B in the right cluster. Intuitively, this step will automatically determine the number of errors present in the design, which corresponds to the number of clusters. Once the center of each cluster is determined, the Euclidean distance D_j^i between each suspect $s_j^i, 1 < j < |S_i|$ in set S_i, for all suspect sets, and the center of its cluster is calculated. This distance will be used in the final weighted ranking step.

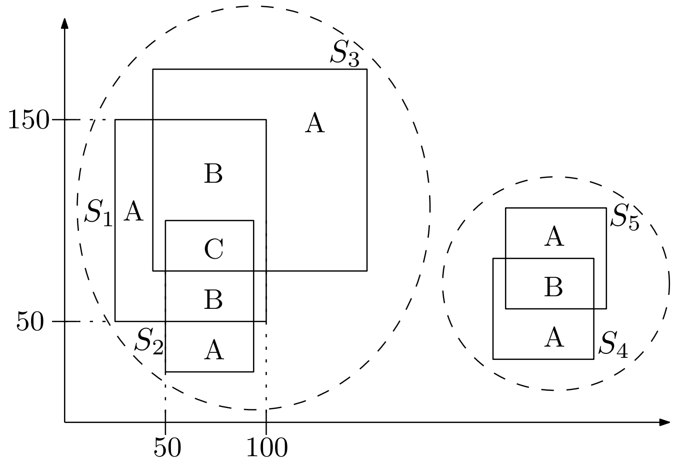

Fig. 1. Clustering-based revision debug: clustering step

Revision and branch classification is performed next. It will be illustrated here for revision classification, but can be extended for branch classification in a straightforward manner. A Support Vector Machine (SVM) [10] is trained on commit messages from past revisions in the VCS. Unique words in the commit message are used as features. Once trained, the SVM can be used to predict the probability that a revision R_k is a bugfix or not, P_k, based on its commit message. The intuition behind this step is that revisions that are bugfixes are less likely to introduce an error into the design. Alternatively, this step can be replaced by extracting the bugfix status of a revision or branch directly from the ITS, if available.

Once clustering and classification are finished, weighted ranking can be used to rank revisions (and branches). The following formula is used to calculate weights for each revision R_k:

$$w_k = min_{i,j}\Big(\frac{1}{2}\Big(\frac{D_j^i}{max_{i,j}(D_j^i)} + P_k\Big)\Big) \qquad \forall i,j | s_j^i \in R_k \tag{1}$$

Intuitively, a weight is assigned to each revision based on the distances calculated in the clustering step, and the probability the revision is a bugfix, determined in the classification step. The smaller the weight assigned to a revision, the more likely it has introduced an error when compared to other revisions.

Finally, the revisions are sorted by weights and separated by cluster. The lists of revisions for each cluster are then merged, with equally-positioned revisions being assigned an equal rank. This is exampled as follow, where the revisions in cluster C_1 and C_2 and ranked in the unified list C':

$$C_1 = \begin{pmatrix} R_1 \\ R_2 \\ R_4 \\ \dots \end{pmatrix}, C_2 = \begin{pmatrix} R_1 \\ R_3 \\ R_4 \\ \dots \end{pmatrix}, C' = \begin{pmatrix} R_1 \\ R_2, R_3 \\ R_4 \\ \dots \end{pmatrix} \tag{2}$$

The above methodology has a major shortcoming in that it does not provide an absolute probability that a revision has inserted an error. The weight of each revision is relative, and the final ranking provides no means of determining the distance between the ranks.

D. *Logistic Regression*

Perceptrons based on logistic regression are equivalent to single-layered artificial neural networks. For an input X, the output y follows the general form:

$$y = \frac{1}{1 + e^{-f(X)}} \tag{3}$$

with $f(X)$ being an analytic function in X. As the function is continuously differentiable, it can be used for backpropagation (*i.e.,* training). In addition to being simple to use, this classifier is chosen as the output is continuous, allowing for a real-valued probability to be predicted. While perceptrons based on logistic regression can be used as multiclass classifiers, for the purposes of this work they are used as simple binary linear classifiers.

III. Perceptron-Based Revision Debug

This section presents a novel perceptron-based revision debug framework using logistic regression [11]. A summary of the flow is outlined below, followed by detailed explanation of each step.

A. *Overall Flow*

The overall flow of this methodology is shown in Fig. 2. This methodology differs from the clustering-based methodology of [4] in that, at its core, it uses a perceptron to rank revisions rather than a clustering algorithm. Perceptrons have the advantage of being extensible with additional features, and can be fine-tuned through training and adjusting of hyperparameters. The methodology begins as follows: revisions from the VCS are *flattened* from the typical branching format to a one-dimensional list. This allows revisions to be sent as inputs to the perceptron naturally. This data is merged with suspects from an automated debugging tool.

Training is performed by manually labeling a collection of the revisions as being the root cause of a failure (*i.e.,* having inserted an error into the design) or not. The list of labeled revisions is used as input to the perceptron. Finally, prediction is performed, returning the real-valued probability that a revision has inserted an error into the design.

B. *Revision History Formatting*

The first step to training the perceptron involves generating a list of flattened revisions. The branching nature of modern VCSs must be represented in linear form, in a format suitable to be used as input to a perceptron. Two methods of flattening the revision history graph are presented here: a simple revision-to-revision method and a more involved revision-to-head method. For the latter method, revisions will retain a `diff` (*i.e.,* list of changes) to a head, a failing revision. This allows for a more direct matching to suspects, but can result in more false positives, in contrast to the former method, as revision functionality may be masked or deleted over the course of the design's history.

For the first method, revision-to-revision, the raw `diff` of each revision is acquired. This `diff` is then parsed to extract the lines that have been modified, which are tagged to the revision.

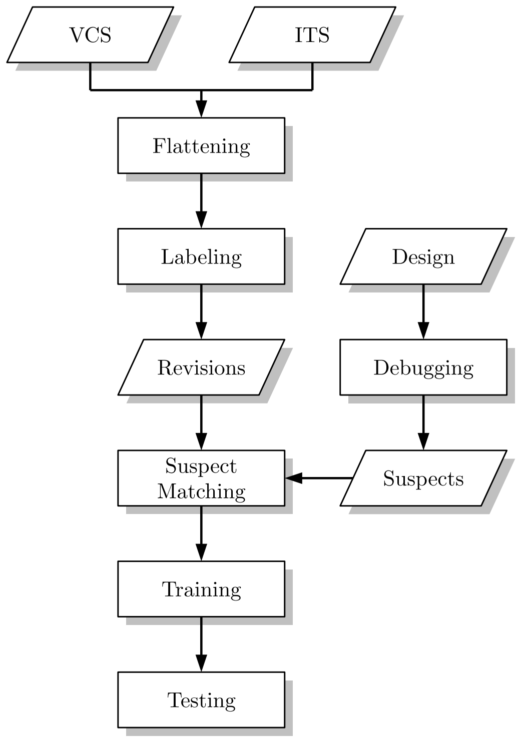

Fig. 2. Perceptron-based revision debug flow

The second method involves revision-to-head `diff` operations. By modeling the revision history of the design as a graph, a Depth-First Search (DFS) is performed starting at each head, going backwards through the revision history. The policy of this DFS is to visit child branches first, before returning to the parent branch. At each revision visited, a `diff` between the revision and the head is generated, which is then compared to the `diff` of the previous revision. The difference between the two sets of lines changed corresponds to the set of lines changed in the current state of the design by the visited revision. This is in contrast to the revision-to-revision `diffs`, which show how a particular revision affected the previous state of the design. As before, revisions are tagged with the set of modified lines. For lines that have been removed from the head, revisions are tagged with half-lines *e.g.,* if a line(s) between lines 4 and 5 of the head have been deleted by the visited revision, it is tagged with a line 4.5. An example input and output of this second method are shown in Fig. 3, in which a revision history with one branch is flattened into a single list of revisions.

Regardless of the process used to flatten the revision history, each revision is additionally tagged with its unique ID, generated from the VCS. A unique branch ID can be assigned to each branch, and revisions are similarly tagged with this ID as well. Using information from the ITS, each branch can be associated with being a bugfix or not.

Once revisions have been flattened, the labeling process, which will be used for training, can begin. A selection of failing revisions throughout the design's history are designated as *heads*. Practical considerations on how heads can be selected are discussed in Section III-D. Each head will serve as the root revision from which previous revisions will be labeled. For each head, a small set of

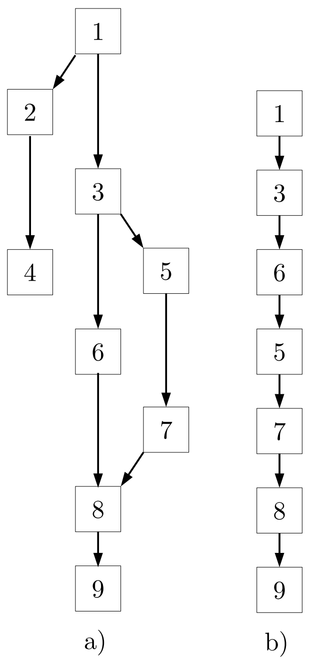

Fig. 3. Flattening revision history

randomly selected revisions, biased towards more recent revisions, are labeled as not being the root cause of the failures observed at the head. Following that, revision(s) that are the root cause(s) are labeled as such.

At this stage, each revision is tagged with its unique ID, the unique ID of its branch, a list of modified lines, and whether it is from a bugfix branch or not.

C. Perceptron Training

In order to train the perceptron, the data gathered from the VCS and ITS must be combined with results from a SAT-based automated debugging tool. For each head, selected previously, the debugging tool is run using the set of failures present, F. For each failure f_i in F, a set of suspects S_i is returned, with each suspect $s^i_j, 1 < j < |S_i|$ corresponding to a range of lines in source files. These suspects will later be matched with revisions, but for now the formatting of input data to the perceptron will be discussed.

In order for the gathered data to be passed to a perceptron, it must be formatted as a set of training samples, each with a fixed number of features. For each head, a random selection of revisions will be used as a training sample, along with the revisions that are known to have inserted an error. Each revision will take on the following features: unique revision ID, unique branch ID, whether it is from a bugfix branch or not, and finally a mapping between suspects and the revision.

Revision and branch IDs are positive integers, and can be generated during the graph traversal of the revision history. The branch bugfix status is a Boolean value, generated by reading the bugfix status of the revision's branch from the ITS.

To format suspect information, they must be matched with lines for each revision that will be used to train the perceptron. For each head, each revision R_k is examined. For each revision, the set of changed lines (either revision-to-revision or revision-to-head) is compared to the set of suspects gathered for the failures seen at the head. For each changed line l that matches a with suspect line, the *matching value* V^k_l incremented. This value represents how closely a changed line matches with the suspects. Intuitively, if a revision makes a change at a line that is a suspect, it is more likely to have introduced an error. When using revision-to-head changes, suspects that match lines surrounding half-lines increment the matching value of the half-line. Intuitively, if a revision makes a change close to a line that is a suspect, it may have introduced an error. The matching value is incremented at a reduced rate however, as the suspect doesn't match the changed line exactly. More formally, this can be expressed as follows:

$$V^k_l = \sum_{s^i_j \in S_i} M(l, s^i_j) + f_m \cdot \sum_{s^i_j \in S_i} m(l, s^i_j) \qquad \forall 0 < i \leq |F| \tag{4}$$

where M and m are boolean functions that check for exact matching and half-line matching between changed line c and suspect s^i_j, as described above, respectively. The constant $0 < f_m < 1$ reduces the impact of half-line matching, and can be tuned experimentally.

Now that the relationships between suspects and revisions has been gathered, they can be formatted to be passed to the perceptron. A feature is used for each unique pair of {design file, line} that has a matching count greater than 0 across all training samples.

More formally, the input vector X_k for revision R_k can be described as the following sparse vector:

$$\begin{aligned} X_k = \Big[& revision\_id, \\ & branch\_id, \\ & is\_bugfix, \\ & \{l : V^k_l\} \forall l | k, V^k_l > 0.0 \Big] \end{aligned} \tag{5}$$

where $revision\_id$ and $branch\_id$ are positive integers corresponding to the unique revision and branch IDs respectively. is_bugfix is a boolean, corresponding to whether the revision is part of a bugfix branch or not. Finally, each non-zero matching value V^k_l for a changed line l in revision R_k, *i.e.,* each line in a revision that matches with at least one suspect, is added to the vector.

The output y of the perceptron is a real value in the range $[0.0, 1.0]$, corresponding to the probability that a revision has inserted an error into the design. The perceptron is trained by labeling samples that are the root cause of a failure with $y = 1.0$, while samples that are not a root cause are labeled with $y = 0.0$.

D. Practical Considerations

This methodology has a key benefit over the previous clustering-based methodology from a practical standpoint: its performance will improve over time. As more training data becomes available, the perceptron's performance is expected to improve. For ongoing designs with a modern workflow consisting of a VCS and ITS, this methodology can be easily integrated. As bugfixes are committed, the source of each bug is known, making it relatively easy to pinpoint the revision that introduced the error. This allows trivial selection of heads on which to train: each time the design experiences a failure which is subsequently fixed, this information can be directly used to train a perceptron.

IV. Performance Extensions

This section presents extensions to the basic logistic regression perceptron introduced in Section III, with the goal of improving classification performance. First, a Support Vector Machine (SVM) is used instead of logistic regression, allowing for non-linear classification through the use of a kernel method. A more advanced method to measure the proximity of suspects to revision changes is also shown.

A. *Support Vector Machine Perceptron*

Using an SVM [10] rather than logistic regression allows for non-linear classification. Intuitively, this means that a perceptron based on SVM can be trained to classify features that are non-linearly separable. This is accomplished through the kernel method [12], in which the features are raised to a higher dimensionality, which can then be linearly separated. The RBF kernel [13], a non-linear kernel, is used for this.

While an SVM requires additional tuning of hyperparameters compared to logistic regression, it can be trained using exactly the same data set, allowing for a seamless transition between the use of the two methods.

B. *Weighted Distance*

Section III-C introduced a basic method of matching suspects with revisions, in which counters are incremented on exact matches. However, there are many cases where suspects cannot reliably matched exactly to changed lines—for example, if a revision deleted a line, it will not be available at the head. This method uses a decaying weight factor to match close-by suspects to revisions.

For each changed line l in a revision R_k, its matching value V_l^k is calculated as follows, with constant f_e:

$$V_l^k = \sum_{s_j^i \in S_i} e^{-f_e \cdot Dist(l, s_j^i)} \qquad \forall 0 < i \leq |F| \tag{6}$$

where distance $Dist$ is calculated as the difference between the line numbers of the changed line l and a suspect s_j^i. If the two lines are in different files, the distance is considered infinite. The matching value V_l^k is the sum of the distances for all suspects found at the head, passed through a decaying exponential function. This allows for suspects that are close to the changes made by a revision, but not exactly matching, to be accounted for.

V. Experimental Results

This section presents experimental results for the novel perceptron-based revision debug framework. The clustering-based revision debug methodology of [4] is compared against variations of the proposed framework. Experiments are conducted on a workstation with an Intel Core i5-3570K CPU clocked at 3.40 GHz, with 16 GB of RAM. A total of nine designs are used, from `OpenCores` [14] and in-house development. Revision histories and issues are gathered from design repositories, which are readily available with the design files. A SAT-based automated debugging tool based on [3] is used to find suspects for the failures at each head. The above data is parsed using a Python platform, which formats it for input to both the clustering-based and perceptron-based systems.

To generate testcases, errors are injected into the latest golden version of the design based on a previously fixed error in the design's history. The pre-existing testbenches are used to observe failures for each testcase, whose erroneous response is recorded for use with the SAT-based automated debugging tool.

Perceptrons, coded in Python, are trained using the above data. From the data, heads are selected based on previous bugfixes (*i.e.,* branches that fix a bug can be used as heads). For each head, the automated debugging tool is ran, then a random selection of between 5 and 10 previous revisions are selected as samples. This selection is biased towards more recent revisions. Once all samples have been selected, additional sample selection passes are ran, with the goal of selecting additional revisions that cover a variety of changed lines. This is needed in case the test revisions contain matching lines that are not present in any of the initially selected revisions. 2-fold cross-validation is used to evaluate the performance of the perceptrons.

TABLE I
Testcase Statistics

Design	Logic Elem.	Num. Heads	Num. Feat. (exact)	Num. Feat. (weighted)	Num. Rev.
6507 CPU	9416	42	131	240	259
ethernet	76408	87	574	992	368
HA1588	9152	26	78	236	70
I2C Core	3640	30	394	673	76
pkt. fwd.	40197	91	50	88	177
SD card	38211	54	262	541	137
SDRAM ctrl	18374	13	974	2109	72
tate pairing	106786	9	83	227	33
VGA	109797	15	115	303	64

Table I summarizes design information. The first two columns show the design name and the total number of logic elements in the synthesized design. Next, the number of selected heads, followed by the number of features for each input sample is shown. Feature count is presented for both the exact matching metric using and weighted distance metric, using revision-to-head `diffs`. Finally, the total number of revisions in the VCS is shown.

Table II compares the ranking results of the clustering-based approach of [4] and three variations of the perceptron-based approach. The first column shows the design name. The next two columns show the ranking of the revision that is used as the injected error base (the *target revision*), and runtime in seconds of [4]. The next three sets of three columns show the revision's ranking, probability of root cause, and runtime in seconds for three variations of the novel framework. First, logistic regression using revision-to-revision `diffs`, followed by logistic regression using revision-to-head `diffs`. Finally, an SVM is used, with revision-to-head `diffs`, and the weighted distance metric shown in Section IV-B.

The benefits of this approach are exemplified across the various testcases. While the average ranking of the target revisions increases by an average of 2.83 times, the average runtime decreases by 75% when comparing the SVM perceptron to clustering. In the case of `6507 CPU`, the ranking improves significantly, due to the target revision being misclassified as a bugfix in the clustering approach. This increases its assigned weight, resulting in an increased rank. In the case of `ethernet`, the relatively large number of training samples available allows the SVM perceptron to be trained to a point that it could surpass the clustering-based approach. It can be generally seen across the table that the SVM perceptron performs better than the logistic regression perceptron, which is to be expected as it has superior classification performance for non-linearly separable features. However, in the case of `SDRAM ctrl`, the SVM performs poorly when compared to the logistic regression. This is because of the weighted distance metric, which causes an increase in the number of input features. In this case, the increase in input features was not offset by a sufficient number of training samples, so the classification performance is lower.

Figure 4 shows learning curves for `ethernet`, using the SVM perceptron. The training and cross-validation scores for a varying number of training samples is plotted. It can be seen that the cross-validation score increases as the number of training samples increases, with a minor penalty to the training performance. Of interest, it can also be seen that the cross-validation score does not converge close to the training score within the maximum number of training samples used. This suggests that the perceptron can be improved if additional training samples are provided. In an industrial setting, this can be achieved easily by integrating this framework with

TABLE II
REVISION RANKING PERFORMANCE

Design	**Clustering [4]**		**Logistic r2r**			**Logistic r2h**			**SVM r2h,weighted**		
	rank	time (s)	rank	y	time (s)	rank	y	time (s)	rank	y	time (s)
6507 CPU	41	2.408	14	0.59	0.530	9	0.64	0.562	5	0.79	0.598
ethernet	6	4.726	26	0.63	0.803	12	0.78	0.817	4	0.87	1.109
HA1588	1	1.091	10	0.74	0.615	11	0.72	0.631	7	0.88	0.822
I2C Core	1	2.165	19	0.63	0.718	16	0.67	0.740	3	0.83	0.972
pkt. fwd.	8	1.153	23	0.68	0.449	18	0.65	0.465	13	0.74	0.503
SD card	4	3.217	11	0.82	0.691	15	0.80	0.705	9	0.72	0.806
SDRAM ctrl	2	25.309	16	0.85	1.207	10	0.89	1.214	25	0.51	1.528
tate pairing	4	0.592	12	0.51	0.617	8	0.55	0.664	8	0.60	0.682
VGA	12	1.384	23	0.49	0.612	19	0.53	0.605	16	0.66	0.699

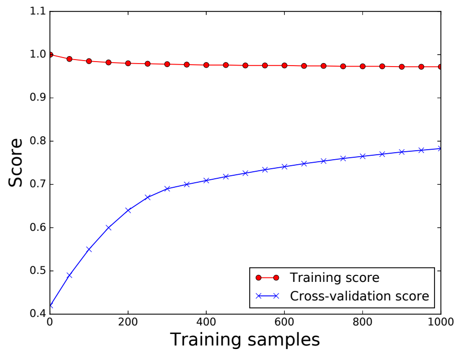

Fig. 4. Learning curves for `ethernet`, using SVM perceptron

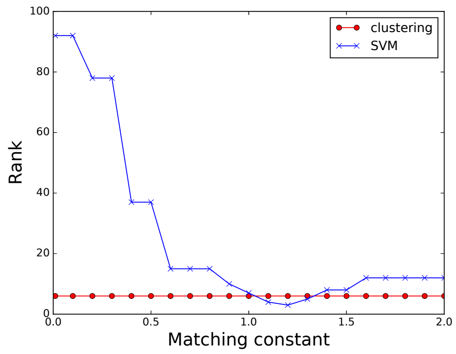

Fig. 5. Target revision ranking with varying f_e for `ethernet`

a workflow making use of a VCS and ITS, as each new bugfix branch can be used as an additional head to train the perceptron.

Figure 5 shows the rank of the target revision for the `ethernet` testcase for varying matching constant f_e, using the SVM perceptron with revision-to-head `diffs` and the weighted distance metric, against the clustering-based approach's ranking. In order to limit the number of features, and overcome the "Curse of Dimensionality" [15] from negatively affecting the perceptron, matching values below 0.1 are discarded. Intuitively, this means that if a suspect is farther than a certain threshold from a changed line in a revision, it is ignored. Ranks closer to 1 are preferred, as the target revision should be prioritized for manual analysis during coarse-grain debugging. It can be seen that for very small and large values of f_e, the ranking of the target revision increases. For the former case, this is because the number of features increases to the point that it is much greater than the number of training samples. In the latter, the weighted distance metric degenerates to the exact matching metric, which does not provide as much information for the perceptron to learn from.

VI. Conclusion

This paper introduces a novel revision debug framework based on logistic regression and SVM perceptrons. This methodology automatically determines the probability that a given revision has introduced an error into the design. It has the advantage over previous clustering-based work in that it is flexible and can be extended with additional features. Extensive experimental results show that it remains competitive with the previously mentioned approach, but has room to improve through additional training. In an industrial context, this framework can be integrated in the workflow to automatically generate new training samples at minimal cost.

References

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

[2] ——, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Intl Conference on Computer-Aided Verification (CAV)*, 2008, pp. 5–10.

[3] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.

[4] D. Maksimovic, A. Veneris, and Z. Poulos, "Clustering-based revision debug in regression verification," in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, Oct 2015.

[5] J. Adler, R. Berryhill, and A. Veneris, "Revision debug with non-linear version history in regression verification," in *1st IEEE International Verification and Security Workshop*, July 2016.

[6] Linus Torvalds, "git, Release 2.8.1." [Online]. Available: https://github.com/git/git

[7] Joel Spolsky, "Painless Bug Tracking." [Online]. Available: http://www.joelonsoftware.com/articles/fog0000000029.html

[8] E. Alpaydin, *Multilayer Perceptrons*. MIT Press, 2014, pp. 640–.

[9] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, 2007. [Online]. Available: http://science.sciencemag.org/content/315/5814/972

[10] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, May 2011. [Online]. Available: http://doi.acm.org.myaccess.library.utoronto.ca/10.1145/1961189.1961199

[11] D. Freedman, *Statistical Models: Theory and Practice*. Cambridge University Press, 2009.

[12] J.-P. Vert, K. Tsuda, and B. Schölkopf, *A primer on kernel methods*. Cambridge, MA: MIT Press, 2004.

[13] A. J. Smola, B. Schölkopf, and K.-R. Müller, "The connection between regularization operators and support vector kernels," *Neural networks*, vol. 11, no. 4, pp. 637–649, 1998.

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

[15] C. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. Springer, 2006.