Clustering-based Revision Debug in Regression Verification

Djordje Maksimovic[1], Andreas Veneris[1,2], Zissis Poulos[1]
[1]Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada
[2]Department of Computer Science, University of Toronto, Toronto, Ontario, Canada
{djordje, veneris, zpoulos}@eecg.toronto.edu

***Abstract*—Modern digital systems are growing in size and complexity, introducing significant organizational and verification challenges in the design cycle. Verification today takes as much as 70% of the design time with debugging being responsible for half of this effort. Automation has mitigated part of the resource-intensive nature of rectifying erroneous designs. Nevertheless, most tools target failures in isolation. Since regression verification can discover myriads of failures in one run, automation is also required to guide an engineer to rank them and expedite debugging. To address this growing regression pain, this paper presents a framework that utilizes traditional machine learning techniques along with historical data in version control systems and the results of functional debugging. Its aim is to rank revisions based on their likelihood of being responsible for a particular failure. Ranking prioritizes revisions that ought to be targeted first, and therefore it speeds-up the localization of the error source. This effectively reduces the number of debug iterations. Experiments on industrial designs demonstrate a 68% improvement in the ranking of actual erroneous revisions versus the ranking obtained through existing industrial methodologies. This benefit arrives with negligible run-time overhead.**

I. Introduction

Verification has become a major challenge in the industrial design process, taking as much as 70% of the design cycle [1]. Half of this time is spent in debugging ensuring that functional errors do not escape to the chip manufacturing stage where the costs will be steep and may jeopardize the time-to-market constraints. Despite of its importance, debugging remains a predominantly manual process.

Verification is performed either *on-line* or *off-line*. On-line verification is the task where an engineer analyzes the design through model checking or simulation to discover an error trace(s) that exposes a particular failure. This is followed by functional fine-grain debug which aims to localize the source of the failure. Today, on-line debugging has been automated with tools such as those in [2]. They use the error trace(s) to locate possible sources in the design Register Transfer Level (RTL) that are responsible for the failure. Next, these sources are mapped back to the Hardware Description Language (HDL) design representation and they are presented to the engineer.

On the other hand, regression (or off-line) verification runs extensive test suites, usually overnight, to exercise a majority of the design functionality. When the verification engineer examines the results of regression the next day, debugging is usually performed in a coarse-grain manner by parsing simulation logs and error messages. Candidate errors are discovered and distributed to the appropriate verification or design engineer for a more detailed fine-grain analysis. Due to the excessive amount of information that needs to be analyzed after regression, and because engineers working on the same design can be geographically and time-zone dispersed, it is not a coincidence today that industry declares that for every single designer there are three verification engineers [1]. As such, if course-grain debugging is not accurate, this may result in significant delays because the wrong debug case may be passed to the engineer, only for them to examine it and return it back, thus increasing the number of debug iterations.

To improve regression verification and the subsequent debugging process, the data returned must be preprocessed in a way that can provide confidence towards the cause of the error. Towards this direction, recent work in *failure triage* [3] has improved the process by implementing machine learning engines that determine which engineers are best suited to rectify a failure. Although useful, this work tries to cluster (*i.e.,* bin) failures according to their impact in the functionality of the design. In other words, it does not take into consideration other useful sources of information such as version control systems. These systems indicate what type of changes were implemented at the particular stage of the design process after the last successful verification step. Evidently, if this information is analyzed accurately, it can provide useful rankings for the candidate source of error(s) and aid the subsequent debugging process. Although failure triage shares similar objectives, by definition its nature is complementary to the work presented here.

This paper introduces a novel approach, which utilizes automated debug and existing information from version control systems, to rank design revisions using traditional machine learning techniques. The ranked revisions aid in determining which error source, originally identified by a debugger, is the actual error source that is responsible for a particular failure following regression. The prioritized revision(s), along with the failure(s), are sent to the design engineer(s) that committed the revision(s) to give them an intuition of what is causing the failure and when it was introduced. This precise step of analysis aims to ultimately reduce the number of debug iterations. The algorithm applies Support Vector Machine classification [4] to first identify which revisions are bug fixes and which are incremental improvements. Then, a functional debugging step is performed and followed by Affinity Propagation [5] clustering to rank potential error sources. A feature of this clustering algorithm is the ability to distinguish between failures caused by the same source and failures originating from different sources. Finally, a weighing scheme is utilized to identify the revisions which are most likely to be the source of the observed failure.

There are several additional benefits of using this algorithm in the verification flow. First, if a design previously passed verification but a new revision broke its functionality, the algorithm will identify the revision responsible for the failure. Next, as empirical results demonstrate, the intelligent parsing of the information as proposed here, triumphs over brute force script-based ranking approaches that are traditionally used in the industry. Specifically, when the brute force assigns a rank to the actual erroneous revisions, the ranking achieved by the algorithm presented here is on average 68% better than the brute force ranking. This arrives with an average run-time overhead of merely 4.631 seconds.

The remainder of this paper is outlined as follows. Section II contains a presentation on related prior art, background material on functional automated debugging and a description of a brute force script-based approach which draws comparisons with the algorithm presented in the paper. Section III presents an illustrative discussion of the flow and intuition behind the algorithm. Section IV gives a detailed breakdown of the revision debug algorithm. Finally, Section V presents the experiments and Section VI concludes this work.

II. Preliminaries

A. Prior Art

There has been significant research in software verification that uses machine learning on version control systems to assist debugging.

The authors of [6] propose an algorithm for predicting future code changes. Their algorithm analyzes recent revisions and predicts in which file a code change may occur. In [7], the authors propose an algorithm that can detect code patterns in version control systems. This allows them to identify potential bad coding styles. In the work of [8], the authors use Bayesian Belief Networks to predict how a code change will propagate to other modules in the software system. Finally, [9] applies machine learning on version control systems to prioritize code review. The intuitive observation there is that code which people have struggled with in the past is more likely to be bug ridden in the future.

There is much less work in regression debugging for hardware systems. To the best of our knowledge, the only related work is this of [3] that implements a clustering-based failure triage engine to bin the errors after regression and help determine which engineers are best suited to rectify a failure. The authors introduce a data weighing scheme for simulation results, combined with modeling failures as multi-dimensional objects. Failure triage is performed by computing failure relations as distance metrics.

The above work applies machine learning algorithms to the results from traditional functional debug for regression verification. In contrast, in this work the information from version control systems is utilized to improve the overall process of coarse-grain debugging. Although our work presents a method to accelerate revision debug, it does not provide failure binning. As such, the work of [3] is complementary to the one presented here.

B. Functional Debugging

Consider an erroneous design with multiple faults in the RTL. When a mismatch in values between a golden specification and the observed vectors is identified, a failure f_i has occurred. Let the set of multiple failures $F = \{f_1, f_2, \ldots, f_{|F|}\}$ represent a set of failures returned by a regression test run, each originating from possibly different erroneous locations in the RTL.

Recently, SAT-based debugging [2] tools have been developed to automate the task. These tools receive an error trace that exposes failure f_i and they generate a set of candidate lines, denoted as $S_i = \{s_i^1, s_i^2, \ldots, s_i^{|S_i|}\}$, in the HDL description of the circuit, each of which can be modified to eliminate the faulty response for the particular error trace. Set S_i is an over-approximation in the sense that it is guaranteed to contain the actual error location but it may also contain other locations that cannot be corrected for all input test vectors. In other words, traditional debugging does not rank the elements in set S_i and therefore, an engineer may spend a substantial amount of manual labor to identify the actual error location when the automated debugger returns many potential error sources.

C. Brute Force Approach

The following brute force approach is used to compare metrics of the algorithm presented later in the paper. This approach can rectify the problem without using automated debug or machine learning, and is typical of what is happening today in the industry.

When regression returns a set of failures, the brute force approach is executed. Each output error trace is compared to an expected golden trace. For every mismatch, the Breadth First Search algorithm is applied at the mismatching output signal to identify the fan-in cone. Each signal in this cone is then matched to their corresponding definitions in the HDL representation of the design. Finally, each revision in the version control system is parsed to see if they make any changes to these definitions and returned to the engineer if so.

Note that the matching HDL definitions will include the actual error source, as this is an over-approximation. However, it may include locations which cannot mask or fix the bug. Therefore, a percentage of the returned revisions may not be related to the failure.

III. Revision Debug Flow

This section contains an overview of our methodology and it is meant to give an intuitive understanding of the reasoning behind the algorithm presented in detail in the next section.

To improve coarse grain debugging, revisions are ranked based on their likelihood of causing a failure that was identified through regression. Revisions are added to a list in which high ranking revisions appear earlier in the list and low ranking revisions appear near the end of the list. This list, along with the failure(s), is presented to a verification engineer who must distribute the revision(s) and the failure(s) to the design engineers that created them for manual inspection. The design engineer uses the revision(s) to identify whether, when, and why they caused the failure(s). If the modifications applied by the revisions are determined to generate erroneous behavior, then they are corrected. The overall ranking algorithm consists of three distinct phases.

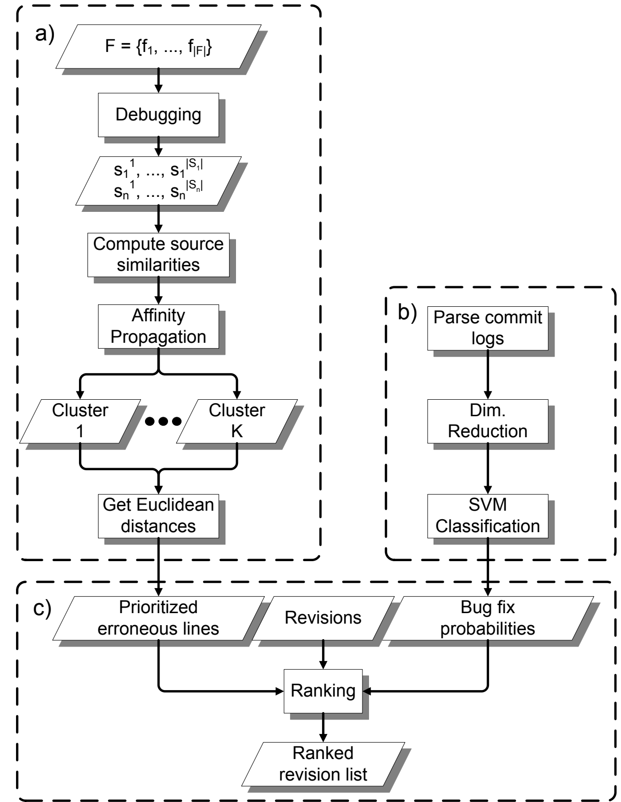

Fig. 1. Flow graph of the proposed revision debug methodology

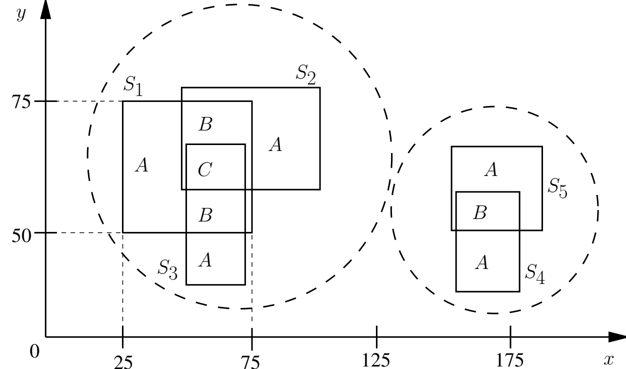

Fig. 2. Clustering to prioritize error sources

In the first phase, presented in Fig. 1(a), the algorithm applies automated debugging [2] to find the potential error sources (*i.e.,* lines of code in a RTL file) of the failures exposed during regression. Following this, the likelihood of each error source being an actual error is determined. For illustrative purposes, Fig. 2 explains this method using potential error source sets $S_1 \ldots S_5$ for five failures $f_1 \ldots f_5$ across two RTL code files. In this figure, each axis represents a RTL code file and the metric on each axis are line numbers in

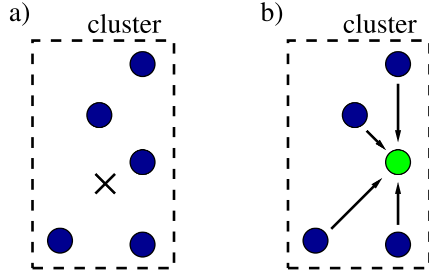

Fig. 3. a) K-Means using a cluster center b) Affinity Propagation clustering

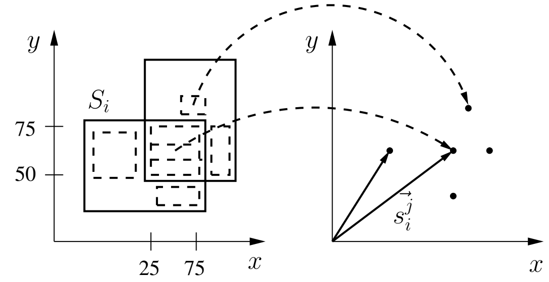

Fig. 4. Converting an area graph to a point graph

the file, *i.e.,* coordinate `(1,1)` is the first line in each file and the coordinates extend past the end of the files.

For each failure, the set of error sources which can correct the failure are mapped onto the graph, as seen in Fig. 2, as solid boxes to represent locations for possible correction. For example, assume that S_1 in Fig. 2 has two potential error sources in two files (represented by the x and y axis). One error spans lines `25-75` of the file along x and the other spans lines `50-75` of the file along y. Hence, the area that represents S_1 in the graph would span coordinates from `(25,50)` to `(75,75)`.

Intuitively, when two or more areas overlap, it indicates that the failures may be the result of the same error source because debugging is indicating similar RTL code for correction. In Fig. 2 there are three different types of areas, which are denoted as `A`, `B` and `C`. Area `A` has no overlap, `B` is an overlap between two regions and `C` is an overlap between three. Area `C` represents RTL code that can be corrected to fix failures f_1, f_2 and f_3, while f_4 and f_5 can be corrected by modifying the right-most area `B`. For the sake of simplicity, if an engineer were to correct the failures, they would target RTL code which when corrected eliminates the most failures. Therefore, code which overlaps has higher priority for correction. In the figure, priority is given to code in overlap `C`, followed by `B` and then by `A`. In essence, the purpose of clustering is to identify the regions that have the greatest overlap. In the figure, the dashed circles represent a cluster and the number of clusters indicate the minimum number of actual errors in the design. Revisions that make changes to code in the overlap are given a higher ranking.

The second phase of the algorithm, illustrated in Fig. 1(b), utilizes commit logs from version control systems to classify revisions as either bug fixes or not. One usually determines whether a revision is a bug fix by interpreting keywords from the commit log such as "fix." Thus, a classifier can classify revisions by utilizing unique words from commit logs as attributes for determining the classification. The intuition here is that a bug fix revision has a lower chance of being buggy when compared to revisions that make other changes. Thus, if a revision is a bug fix it would receive a lower ranking.

In the final phase, demonstrated by Fig. 1(c), revisions are ranked according to their likelihood of causing the observed failures using data gathered from the previous phases. Revisions that are not related to the failures are not included in the list. The following section presents the details of each process in Fig. 1, respectively.

IV. Generating the Ranked Revision List

A. Error Source Clustering

When regression is executed, each test exercises a portion of the design functionality. Failures may appear different because they can be detected through multiple mechanisms, such as assertions or a mismatch, as identified by a checker, between the observed and expected values. Despite this, they may have originated from the same source. The debugging results (potential error sources) for failures originating from the same error source will be similar. This is in contrast to failures originating from different sources which will have less comparable debugging results. The concept addressed here locates the similarities between failures so they can be used to aid in distinguishing between an actual error source and false positives.

Clusters represent the perceived number of actual errors in the design. Since the number of real errors is not known, an algorithm is needed which does not require the number of clusters K to be known a priori. Affinity Propagation (AP) [5] is a clustering algorithm which can perform this task in the metric space. In traditional algorithms such as K-means, as demonstrated in Fig. 3(a), a priori knowledge of the number of clusters K is needed because the objective is to minimize the distance between members and the K cluster centers (denoted by `X` in the figure). In AP, clusters are created by maximizing the similarities between points and selecting an exemplar, a point which most closely represents the cluster. This is seen in Fig. 3(b) where members point towards the exemplar.

As discussed before, the potential error sources for each failure span an area in a space where dimensions are lines of an RTL file. However, the AP algorithm works with points in a space, therefore the data is preprocessed as illustrated by Fig. 4. In this figure, the solid boxes denote the area covered by all the sources for a particular failure combined (S_i) and dashed boxes indicate a subset of this area. The dashed arrows demonstrate how a single source s_i^j is converted to a point in space. The source is a tuple $(startLine_i^j, endLine_i^j, file_i^j)$, where $startLine_i^j$ ($endLine_i^j$) denotes the start (end) line of the possibly erroneous code in the file $file_i^j$. For each s_i^j a vector $\vec{s_i^j}$ is created by assigning the axis spanned by s_i^j to $(startLine_i^j + endLine_i^j)/2$. Every other axis (where $axisName$ is the name of the file along that axis) is assigned as follows:

$$average\left(\frac{startLine_{i'}^{j'} + endLine_{i'}^{j'}}{2}\right), \quad \forall\ i', j' \mid axisName \in s_{i'}^{j'} \wedge s_{i'}^{j'} \in S_i \tag{1}$$

i.e., the average of each $s_{i'}^{j'}$ in S_i spanning the other axes. In essence, this ensures that vectors will be located around regions with overlap, with exemplars being those vectors located within the overlap.

Example: Suppose s_i^j spans lines `25-75` on the x axis and two other sources span lines `50-65` and `60-75` on the y axis. To convert the area of s_i^j to a point, the co-ordinate is given as `((75+25)/2, ((65+50)/2 + (60+75)/2)/2)`, *i.e.,* $\vec{s_i^j}$ is `(50, 62.5)`.

The AP algorithm determines clusters by maximizing the similarities between points in the cluster. The negative squared Euclidean distance similarity metric is applied, which allows the creation of clusters from those vectors that are in proximity to each other. Assuming T is the total number of error sources returned by debugging, the similarity $sm(x_1, x_2)$ between two points is computed as follows:

$$sm(x_1, x_2) = -||x_1 - x_2||^2 \tag{2}$$

where x_1 and x_2 are function parameters. This function is used to create a $T \times T$ similarity matrix which calculates the negative squared Euclidean distance between $\vec{s_i^j}$ and every other point. Each vector has a preference associated with it that determines the probability of it being an exemplar. The preference for a vector is defined as the median of the similarities between that point and every other point.

Following the AP algorithm execution, the Euclidean distance between the cluster exemplar and the source is computed as follows:

$$D(i,j) = ||exmp(label(\vec{s_i^j})) - \vec{s_i^j}|| \tag{3}$$

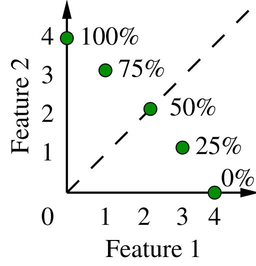

Fig. 5. Illustration of Support Vector Machine classification

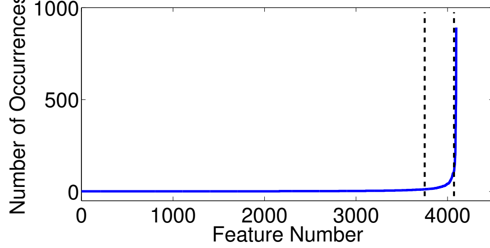

Fig. 6. Word occurrences in a feature list from eleven OpenCores designs

where $exmp(x)$ is a function that takes a cluster label and returns the exemplar, and $label(x)$ is a function that takes a vector and returns the associated cluster label determined by AP.

From the way the vector graph is constructed, the exemplar will be in an area with overlap. Since those areas closely represent the code which can be used to fix multiple failures in the cluster. Then, the closer another point is to the exemplar, the more likely it is to correct multiple failures. As such, the smaller $D(i, j)$ is the more it influences a revision to receive a higher ranking.

B. Revision Classification

To promote organization, engineers frequently employ version control systems that manage changes (*i.e.,* revisions) to documents, computer programs and other pertinent information. Each revision usually targets information for a specific task. For example, a revision can be classified as adding functionality, modifying functionality, removing functionality, bug fixing, etc. Since commit logs describe what changes an engineer has made in that revision, then it can be classified by applying its commit log to a machine learning classifier.

Revisions are classified through the use of Support Vector Machines [10] (SVM). An SVM is an algorithm which predicts a classification of an object based on certain properties (*features*). For the methodology in this paper, features are unique words in a commit log and revisions are classified as either a bug fix or not. Fig. 5 shows an SVM classification with two features. The dashed line is the classification boundary, indicating that `Feature 2` is a word related to bug fixing (such as "bug") and `Feature 1` is a word unrelated to bug fixing (such as "implemented"). Each axis indicates the amount of occurrences of each feature within a commit log. Dots represent a classification of a revision with probabilities ranging from `100%` bug fix to `0%` bug fix (*i.e.,* not likely bug fixes).

SVMs are chosen for two major reasons. First, SVMs can represent non-linear classifications through the use of kernel functions [4] by mapping data into high dimensional feature spaces. Second, SVMs are capable of representing predicted classifications as probabilities. This aids in achieving a uniform ranking in cases where a revision includes both bug fixes and incremental upgrades.

The classifier is taught to classify revisions through examples of revisions that are bug fixes and ones that are not. Once trained, it develops a prediction model that it uses to predict future classifications. For a set of N commit logs, each is manually labeled as either bug fix or not, to construct a training label set $L = \{l_1, \ldots, l_N\}$. Following this, the number of M unique words in the N commit logs are used to construct a data matrix $C_{n,m}$ for $1 \leq n \leq N$ and $1 \leq m \leq M$, where each element is an integer counting the amount of times word j appears in commit log i. In this sense the labels and data matrix indicate examples of revision classifications. Once initial training has completed, any subsequent classifications only require the prediction model and the features of a revision commit log.

A critical inefficiency of SVMs is that they provide poor classification accuracy if $M >> N$ [4]. This is widely known as the "Curse of Dimensionality" where the number of dimensions exceed the number of samples. In the data matrix, it is undoubtedly true that $M >> N$, because commit logs can be of substantial variable length and word variety. Therefore, dimensionality reduction is applied to reduce the amount of unique words trained on, by removing words that appear too commonly or too infrequently in the commit logs.

Fig. 6 shows the number of occurrences of `4096` words in `2124` commit logs from eleven OpenCores [11] designs. From this graph it can be seen that many words are used infrequently (such as specific email addresses) and only a hand-full that are common (such as "a," "the," "to," etc.). Between the two dashed lines are the features that are retained for classification, while others are removed from consideration. This was determined by positioning each line such that the difference between occurrences of adjacent words within the lines is minimized, while maximizing the median occurrence. The final quantity of features respects the following constraint:

$$M \leq \frac{N}{2} \tag{4}$$

as this ratio gives reasonable prediction results [10].

Once the SVM learns a prediction model, the classification of subsequent revisions can be determined probabilistically. Their features are identified from the commit log and passed, along with the prediction model, to the SVM, which returns the prediction $bugFix_i$, a real-valued probability number between `zero` and `one`.

Revisions which are bug fixes are expected to have a high probability, while those that are not, have low probability. Revisions which mix bug fixes and other activities are expected to fall in-between. It is anticipated that bug fixes do not introduce additional errors when compared to significant code changes. As such, a higher bug fix probability influences the revision to receive a lower ranking.

C. Weighted Revision Ranking

In the final phase, the data gathered from previous steps is applied to rank the revisions and associate each revision with a particular failure. The changes to RTL code made by a revision, denoted by R_l for $1 \leq l \leq N$ where N is the number of revisions, are gathered. Changes made by the revision are matched with the sources of error determined by debugging. If a match is found, then the revision is considered for ranking. The weight w_l of a revision is used to determine the ranking of each revision. It is calculated as follows:

$$w_l = min_{i,j}\left(\frac{1}{2}\left(\frac{D(i,j)}{max_{i,j}(D(i,j))} + bugFix_l\right)\right), \quad \forall\ i,j \mid s_i^j \in R_l \tag{5}$$

This equation states that the lowest weight is given to a revision which is not a bug fix and matches a $\vec{s_i^j}$ closest to the exemplar in the cluster. Thus, the lower the weight of a revision, the higher the rank it receives.

Following this, K-lists of ranked revisions are created, one for each cluster. Each list contains revisions that match the RTL code within the respective cluster. In this sense, each list contains the revisions that when modified can correct the failures in each cluster. The ordering of the lists is determined by sorting each revision by ascending weight.

Finally, the lists are merged together to create a unified list, a process illustrated by the following example. Suppose two lists A and B are generated, which are ranked revisions for two separate clusters. Another list C is the unified list created from A and B. Also suppose each list is populated as follows:

$$A = \begin{pmatrix} R_1 \\ R_2 \\ R_4 \\ \ldots \end{pmatrix}, B = \begin{pmatrix} R_1 \\ R_3 \\ R_4 \\ \ldots \end{pmatrix}, C = \begin{pmatrix} R_1 \\ R_2, R_3 \\ R_4 \\ \ldots \end{pmatrix} \tag{6}$$

TABLE I
BENCHMARK STATISTICS

Design	Test Num.	Logic Elem.	Num. Err.	Num. Fail.	$\sum_{i=1}^{\lvert F \rvert} \lvert S_i \rvert$	Num. Rev.
Ethernet	1	76408	6	10	589	332
	2	76408	1	4	880	332
HA1588	3	9152	4	6	129	70
	4	9152	3	6	69	70
	5	9152	7	12	198	70
I2C Core	6	3640	3	4	367	70
	7	3640	3	12	178	70
Tate Pairing	8	106786	4	4	159	33
	9	106786	5	37	81	33
SD Card	10	38211	1	20	241	127
	11	38211	4	36	179	127
SDR CTRL	12	18374	2	5	1726	72
	13	18374	8	10	1201	72
6507 CPU	14	9416	2	3	144	259
	15	9416	2	3	75	259
VGA	16	109797	3	5	173	59
In-house Packet Fwd.	17	40197	2	16	42	177
	18	40197	4	23	51	177

Observing list C, it can be seen that R_1 has the highest ranking and makes changes to code in both clusters *i.e.,* it appears that it is responsible for all the errors. If an engineer determines that it is not the cause of the failures, then both R_2 and R_3 are equally considered. This happens because neither revision can be modified independently to correct all the failures. Thus, the ranking indicates that both revisions are likely erroneous and must be corrected.

The final unified list is presented to the verification engineer, who must manually inspect the revisions. It should be observed that due to the inclusion of all existing revisions and the completeness of automated debugging, at least one of the revisions in the unified list will be the cause of the failure(s).

V. EXPERIMENTAL RESULTS

This section presents experimental results for the proposed revision debug framework. Experiments are conducted on a core i5-3570K workbench clocked at 3.40 GHz with 8 GB of RAM. Eight designs from OpenCores [11] and one in-house real-life industrial design are used to evaluate the methodology. Revision changes and commit logs are gathered from the OpenCores Subversion repositories, and are readily available with the design files. A total of eighteen tests are performed. The SAT-based automated debugger used to generate potential error sources for failures is implemented based on [2]. A platform is coded in Python that is used to parse debugging results and generate clusters with the AP algorithm. Classification predictions, for the experimental revisions, are performed without a priori knowledge of whether the revision is a bug fix or not. This is done by training a SVM prediction model prior to conducting the experiments. Finally, the revisions are ranked for each test.

Errors are injected into the design by reading the commit log of the design and identifying actual bugs that the designer had to correct. A portion of these bugs are reintroduced into the design to test the methodology. For each regression run, a preexisting simulation test-bench is used to evaluate the functionality. Each regression run involves hundreds of input vectors. Erroneous responses are detected by comparing observed results to expected results from a "golden" model checker. A separate script is coded to train the SVM and generate a prediction model. This is achieved by parsing revision commit logs and performing dimensionality reduction to overcome the "Curse of Dimensionality." The features that are utilized for prediction model training are also used during the classification of experimental revisions.

Table I summarizes design information and statistics for each experimental run. Columns of this table show the design used for testing, an enumeration of the test runs, the number of logic elements in the design, the number of errors injected in the benchmark, the number of failures observed, the total number of error sources generated by automated debugging and the number of revisions stored by the version control system. The effectiveness of the methodology is evaluated by comparing the ranking results of the methodology with the ranking obtained from the brute force approach described in Section II-C. The comparison is performed by determining which approach gives a higher ranking to the revision(s) that are responsible for the actual errors in the designs.

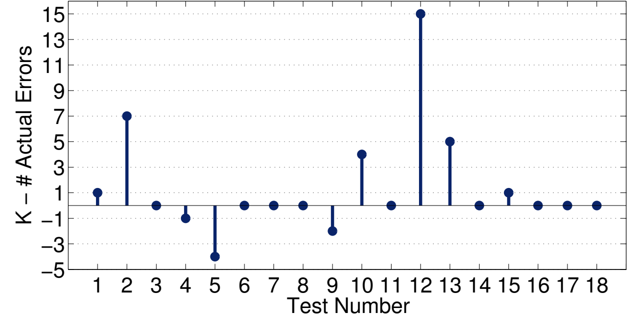

Fig. 7. Cluster prediction error

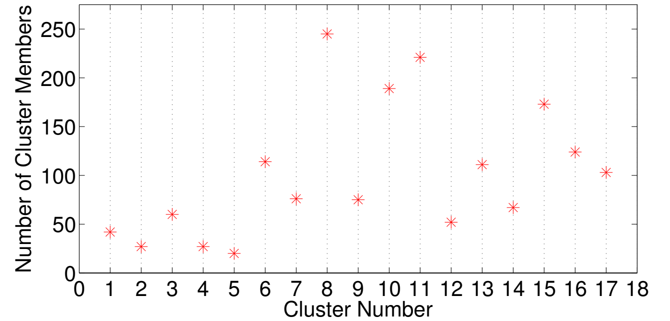

Fig. 8. Number of members in each cluster for `SDR CTRL (test 12)`

When debugging returns a set of potential error sources, these sources are clustered. Fig. 7 shows the prediction accuracy of the AP clustering algorithm. The number of actual errors in the test are compared with the number of clusters. The prediction error is given as K `- # actual errors` for each test case. Ideally, the number of clusters will equal the number of actual errors and the prediction error will be 0. It can be seen that 9/18 tests have perfect cluster prediction, with an average of 4 cluster difference between the remaining tests. Observing test `12`, it can be seen that 15 more clusters are created than needed. This is due to locating 1726 potential error sources across 9 different files, indicating that the two actual errors in this design were difficult to locate. Despite of this error, the exemplars for each cluster provided adequate information as a 2/12 ranking was achieved for the revisions responsible for the errors. The reason for this is illustrated in Fig. 8, which shows the number of members in each of the 17 clusters of test `12`. From this figure, it can be observed that `clusters 8,10,11`, and `15` are the primary influences for the ranking and may contain the actual errors in the design. The remaining clusters contain outliers with little overlap (*i.e.,* false positives).

When training a prediction model, 2124 commit logs from eleven OpenCores design repositories, disjoint from the set used for experiments, are utilized as examples for classification training. From these logs, 4096 unique words are identified. After dimensionality reduction, this set is reduced to 449 words that are used as features for classification. The 2124 commit logs are split into two sets, training and testing, where training receives 70% of the logs and testing receives 30%. The training set is used to create the prediction model, and the testing set is utilized to test the effectiveness of that model. The entire training operation is a one-time process that takes 11.97 seconds, in which dimensionality reduction takes 7.81 seconds and prediction model training takes 4.16 seconds.

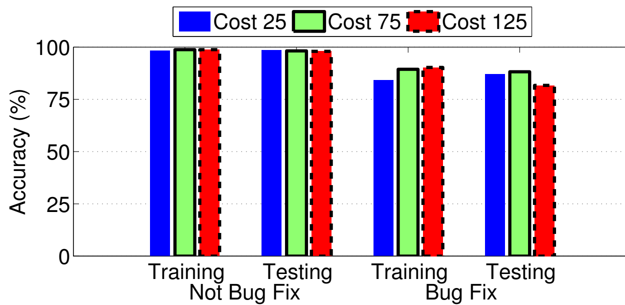

Fig. 9. The classification accuracy for each class with varying cost metrics

Fig. 9 shows the classification accuracy for training and testing revisions. The accuracy metric is computed as the percentage of the number of correct predictions in a class vs. the amount of samples in that class. Combined accuracy (*i.e.,* number of correct predictions with classes combined vs. total amount of samples) was not taken due to the bias towards revisions not being bug fixes. Three different evaluations are performed for three cost settings, where the cost determines the prediction accuracy. A high cost will provide good training accuracy, but poor testing accuracy due to over-fitting, while a small cost will provide poor accuracy. From this figure, it can be seen that a cost of 75 is adequate as the testing accuracy is maximized for this value. There is a prediction accuracy mismatch between classes, due to bug fixes being characterized by a smaller word variety, when compared to other classifications that can be described with a much broader word variety.

Table II shows the results of revision ranking and presents a comparison of the ranking achieved by the proposed methodology vs. the industrial standard (brute force). From left to right, the columns of this table describe the test number, the rank and the time achieved by the brute force method, and the rank and time achieved by the proposed methodology. In the final column, the improvement percentage is given, which indicates how much higher the ranking of the proposed methodology is when compared to the brute force method. This is calculated as $(1-(Proposed\ Rank/Brute\ Rank))*100\%$. From this table it can be seen that on average the proposed methodology achieves a 68.0% higher ranking than the brute force method. This arrives with a total amount of revisions that an engineer may need to look at that is 34.0% of the total amount returned by the brute force method. This particular inefficiency of the brute force method is due to including revisions which are not related to the actual errors.

In Table II, `tests 14-16` give poor ranking results due to a combination of two factors. First, the erroneous revision contains bug fixes as well as an incremental upgrade and second, the upgrade itself introduces new errors. Since the erroneous revision made changes that could have been split up into separate clusters. Then, this results in a classification inaccuracy, which gives the revision a 75% bug fix probability that negatively affects the ranking. Despite of these setbacks, the ranking can be improved by only taking into consideration the results of automated debugging as clustering accuracy is not affected. In some tests, such as `test 5`, extra rankings are given because multiple revisions are erroneous. They make changes to code in similar locations, which means that their error sources appear in the same cluster. This results in each revision receiving a separate ranking.

Regarding run-time concerns, a majority of the time spent is in automated debugging, which takes between 400 to 3600 seconds. This process must always be performed during regression verification, regardless of whether it is manual or automated. In the case of manual debugging, the run-time is exceedingly longer than that of automated debugging. The overhead for performing clustering and ranking is between 0.6 to 30 seconds.

An interesting observation made is that the size of the revision does not dictate whether the revision is more likely to be buggy, or more likely to be an upgrade. In general, including this metric when determining whether a revision is a source of an error does not influence the ranking in any way. Intuitively, this can be explained because some designers debug their code extensively before committing, while others may not.

TABLE II
REVISION RANKING PERFORMANCE

Test Num.	Brute Force		Proposed		improv. (%)
	Rank (rank/total)	Time (s)	Rank (rank/total)	Time (s)	
1	58/129	0.235	6/27	4.335	90
2	84/129	0.244	5/10	6.389	94
3	11/12	0.231	1/5	0.878	91
4	11/26	0.447	1/7	3.322	91
5	15/32 & 22/32	0.24	1/7 & 2/7	3.432	92
6	23/23	0.783	1/13	2.147	96
7	15/23 & 23/23	0.768	1/14 & 3/14	1.931	90
8	16/19	0.087	4/6	0.71	75
9	13/19 & 16/19	0.095	1/3 & 2/3	0.611	90
10	19/32	1.482	4/9	3.213	79
11	19/32	1.445	4/10	3.022	79
12	19/26	1.484	2/12	27.328	89
13	19/26	1.381	2/13	17.976	89
14	6/64	0.183	35/49	2.339	-483
15	27/64	0.173	41/48	2.486	-52
16	3/18	0.305	11/15	1.126	-266
17	17/83	0.166	2/15	0.905	88
18	17/83 & 15/83	0.366	4/16 & 8/16	1.207	62
AVG.	22/47	0.562	7/16	4.63	68

VI. CONCLUSION

Regression verification remains to be a predominantly manual process. To ease verification, this paper introduces a novel clustering-based revision debug framework. The methodology clusters automated debugging results and utilizes version control system data to rank revisions based on their likelihood of being responsible for the failures. Extensive experiments confirm the attractiveness of the approach as they demonstrate improved ranking capability, with negligible run-time overhead, when contrasted to the industrial standard. In the future, we plan to analyze statistical simulation metrics, such as the temporal distance between an error source and the failure observation point, as they may be applicable towards an improved ranking scheme.

REFERENCES

[1] H. Foster, “Assertion-based verification: Industry myths to realities (invited tutorial),” in *Computer Aided Verification*, 2008, pp. 5–10.

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

[3] Z. Poulos and A. Veneris, “Clustering-based failure triage for rtl regression debugging,” in *IEEE Int’l Test Conference*, 2014.

[4] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer Science, 2009.

[5] B. J. Frey and D. Dueck, “Clustering by passing messages between data points,” *Science*, vol. 315, pp. 972–976, 2007.

[6] R. N. A. Ying, G. Murphy and M. Chu-Carroll, “Predicting source code changes by mining change history,” *IEEE Trans. on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[7] B. Livshits and T. Zimmermann, “Dynamine: finding common error patterns by mining software revision histories,” in *European software engineering conference and ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, pp. 296–305.

[8] A. H. S. Mirarab and L. Tahvildari, “Using bayesian belief networks to predict change propagation in software systems,” in *IEEE International Conference on Program Comprehension*, 2007, pp. 177–188.

[9] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, “Bugcache for inspections : Hit or miss?” in *ACM SIGSOFT symposium and the European conference on Foundations of software engineering*, 2011, pp. 322–331.

[10] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, available: http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[11] OpenCores.org, “http://www.opencores.org,” 2015.