Revision Debug with Non-Linear Version History in Regression Verification

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

***Abstract*—Modern digital designs are relentlessly growing in complexity, making their verification a daunting task. Verification and debugging are the bottleneck, accounting for up to 70% of the design cycle. Most automated debugging tools target failures in isolation and rely solely on the current version of a design's RTL. A recently developed methodology targets multiple failures simultaneously while leveraging the revision history present in a version control system. It finds revisions likely to be responsible for the failures and ranks them such that higher ranked revisions are more likely to contain bugs. However, this technique treats the version history as a simple linear list of revisions rather than a graph structure. To address this limitation, this paper presents a technique that properly leverages the branching information in version control systems. It offers two-stage ranking with improved performance, allowing both branches and branch-local revisions to be ranked.**

I. Introduction

In modern hardware design, verification often accounts for up to 70% of the design effort, with 50% of the verification time spent in debugging [1]. This substantial effort is spent to reveal functional errors before chips are manufactured, at which point correcting them is significantly more costly. Due to the extensive engineering resources required by debugging and its importance, automation is necessary to both improve the accuracy of the process and reduce its expense.

Verification can be broadly classified as either *on-line* or *off-line*. On-line verification involves processes such as simulation and formal property checking. From these tasks, an error trace that exposes the failure is readily available and can be used with a Boolean Satisfiability (SAT)-based automated debugging tool [2] to accelerate the fine-grain debugging process. Conversely, it may be found that a particular state is unreachable in violation of the design specification, in which case no error trace is available. An alternative technique can be applied to automate the debugging process [3] for this type of failure. In either case, the tools identify *suspect locations* in a Register Transfer Level (RTL) model of the design that may be responsible for the observed failure. Subsequently, these locations are mapped to lines of Hardware Description Language (HDL) code and presented to the engineer. The set of locations returned is guaranteed to include the actual error source, but may also include several other locations that can merely be used to mask the failure without correcting its root cause.

On the other hand, off-line verification involves large suites of regression tests that exercise a large portion of the design functionality. These tests are often executed overnight or even over the course of several days. Upon completion of the regression suite, verification engineers perform a coarse-grain debugging step in which they analyze the failures that occurred. This may involve parsing simulation logs and error messages in an effort to determine which engineer is responsible for each failing test. Candidate failures are identified and distributed to engineers appropriately for fine-grain debugging. This process is done in an ad hoc fashion and as such it may be inaccurate, resulting in candidate failures being assigned to the wrong engineers, thereby increasing the number of debug iterations and wasting valuable engineering resources.

Recent work in revision debugging [4], leverages information from version control history to rank revisions in order of their likelihood of being responsible for a failure. It utilizes the results from an automated debugging tool, along with existing information from a version control system to generate a list of ranked revisions. Revisions with higher ranking are expected to be responsible for the observed failure. This provides the engineer with valuable information regarding the source of the failure. However, it assumes a linear version control history.

Modern version control systems (VCS) such as Git [5] support complex branching schemes, and as such their history is more akin to a Directed Acyclic Graph (DAG) than a simple linear ordering. To address this limitation, this paper presents an extension to the work of [4] that is able to make better use of the non-linear nature of modern version control systems. In greater detail, sets of revisions corresponding to each branch are classified in order to obtain bug fix probabilities for each branch. The sets are then passed to a weighted ranking system to identify the branches most likely to be the root cause of the failure. Finally, revisions within each set are ranked to provide additional information to the engineer.

This technique has several additional benefits. The reduced number of branches compared to the total number of revisions results in less manual effort required to explore the ranked results. In addition, classification performance is improved when considering branches as a unit. This produces a more accurate final ranking, making it easier for the engineer to pinpoint the branch, and eventually revision, that caused the failure. Experimental results show that branch ranking achieves a 38% better ranking on average, with a 81% increase in average runtime.

The rest of this paper is organized as follows. Section II provides background information on traditional SAT-based debugging and revision debugging techniques. Section III describes the non-linear revision debug methodology. Section IV presents a set of experiments demonstrating the benefits of the proposed methodology. Finally, section V concludes the paper.

II. Preliminaries

A. Version Control Branching

Branching is a commonly-used methodology to isolate code development for a particular feature or bug fix. As exampled in Fig. 1, commits 1, 2, 7 and 11 are on the *mainline*, usually the most up-to-date development version of the code. Other commits are branches. Commits 3 and 6 branch off the mainline while commit 5 branch off another branch (nested branch). Once development on the feature or bug fix is ready, the child branch is merged onto the parent branch,

[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

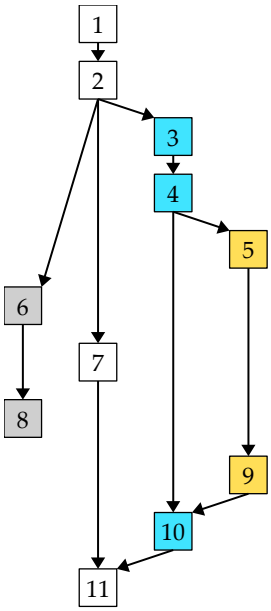

Fig. 1. Revision history as directed acyclic graph

as seen in Fig. 1, commits 10 and 11. When merging, the *diff* of the child branch i.e. the cumulative code changes between the beginning and end of the branch, is applied to the parent branch.

B. Prior Work

In the related field of software verification, significant research has focused on the use of machine learning techniques leveraging version control data. The work of [6] uses machine learning with historical version control data to prioritize files for human inspection. As revisions are created, the system learns which files are likely to contain bugs and therefore should be prioritized for human review. In a similar vein, the work of [7] uses machine learning to identify application-specific coding patterns and styles. Using this information, the framework is able to identify violations of coding practices common to the application, which may be indicative of a bug. Finally, the authors of [8] propose a framework that creates developer-specific prediction models to detect potential defects based on the style and practices of individual developers. To the best of our knowledge, the only technique leveraging version control systems in the field of hardware verification and debugging is that of [4].

C. SAT-Based Debugging

The work of [4] makes extensive use of SAT-based automated debugging [2]. Due to its importance to our work, it is explained here in greater detail. Consider a circuit with one or more faults in the RTL. When regression verification detects a failure due to an observation value mismatch, firing assertion, scoreboard discrepancy, or other similar means, an error trace is returned that exposes the failure. Let $F = \{f_1, f_2, ..., f_{|F|}\}$ denote the failures returned by a regression test run, where different failures may have different underlying root causes.

SAT-based debugging tools [2] can be used to find a set of candidate lines where a fix may be implemented to correct the erroneous behavior exposed by an error trace. For a failure f_i, the tool returns a set of candidate lines $S_i = \{s_{i^1}, s_{i^2}, ..., s_i^{|S|}\}$. Each element of S_i is an entity in the HDL representation of the circuit. It could be a module, or a block, an expression, etc., but ultimately it is mapped to a range of lines of HDL source code. Each such candidate line represents an element in the HDL where a change could be made to correct the erroneous behavior. The technique is exhaustive, so the set S_i is guaranteed to contain the actual error source causing the observed failure f_i. However, in practice it may contain many other locations that can merely be used to mask the failure. As an engineer must investigate entries in S_i to correct the failure, investigating these additional locations may involve substantial engineering effort.

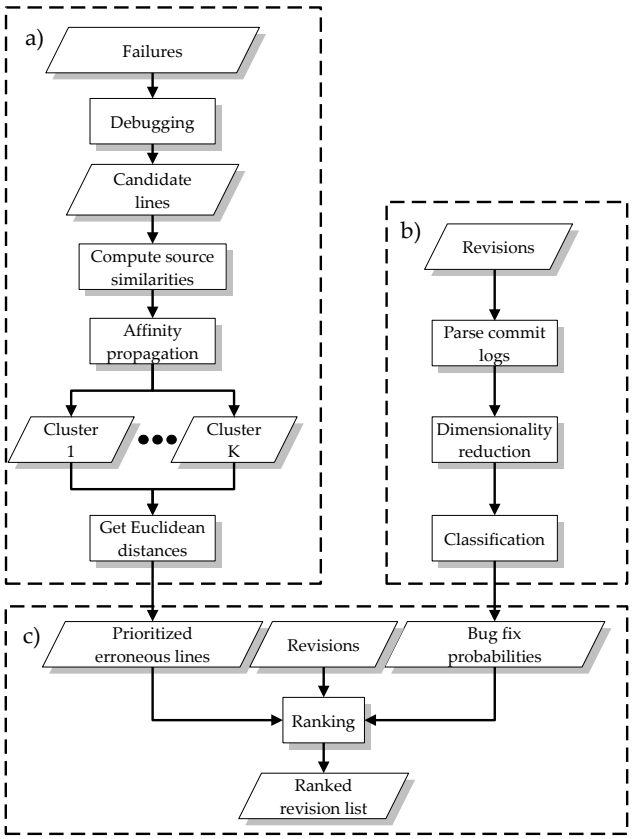

Fig. 2. Outline of linear revision debug flow

D. Revision Debugging

Towards the goal of alleviating the time spent by engineers investigating each candidate line returned by an automated debugging tool, the work of [4] provides a means of ranking them. Given a set of failures, the algorithm returns a ranked list of revisions, where higher-ranked revisions are expected to have a higher likelihood of being responsible for some or all of the observed failures.

In greater detail, the approach works as follows. It is divided into three broad phases, which are shown in Figure 2. The first step, shown in Figure 2(a), uses SAT-based debugging to identify a set of candidate lines for each failure. Since each error in the RTL may be responsible for multiple failures, Affinity Propagation [9] is then used to cluster the failures. The clustering is such that failures in the same cluster may have the same underlying root cause in the RTL, while failures in two different clusters definitely do not have the same root cause. As such, the number of clusters is intuitively equal to the minimum number of errors in the RTL. Additionally, each cluster has an exemplar, which is a line of source in the region of greatest overlap. This exemplar is expected to be close to the actual error source.

Figure 3 shows a sample clustering, demonstrating this concept. In the figure, each axis represents a different HDL source file. Each

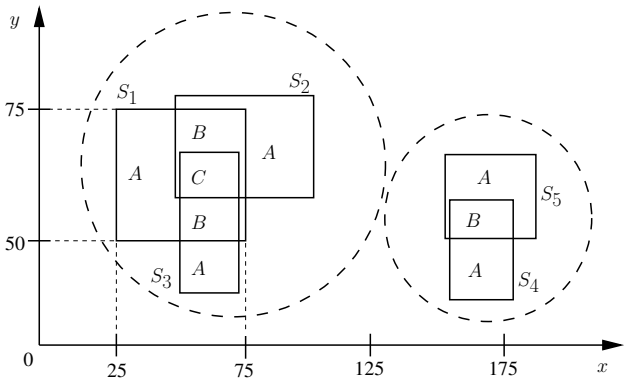

Fig. 3. Example clustering generated during revision debug

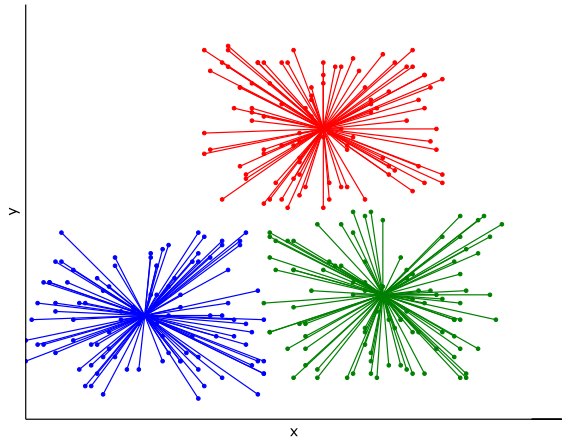

Fig. 4. Affinity propagation clustering

of S_1 through S_5 is the set of candidate lines returned for the corresponding failure. It can be seen that S_1 contains lines `50-75` of the file represented by the y-axis and lines `25-75` of the file represented by the x-axis. The dashed circles represent two clusters, meaning that there are at least two separate errors in the RTL. It can be seen that a correction in the region labeled `C` may be able to correct S_1, S_2, and S_3 simultaneously.

The second phase shown in Figure 2(b) parses the commit logs to determine which commits are bug fixes. Intuitively, it is expected that bug fixes are less likely to introduce bugs than other types of changes are. The results of this step will be used in the ranking phase to reduce the ranking of bug fix revisions. This step begins by parsing and tokenizing the commit logs. The set of N commit messages is converted to an $N \times M$ matrix, where entry i, j for $1 \leq i \leq N$ and $1 \leq j \leq M$ is the number of times word j appears in commit message i. As such, M is the number of unique words in the commit logs. This mirrors how a human would attempt to identify bug fix revisions, by looking for words such as "fix" in the commit log.

After converting the version control history to its matrix representation, dimensionality reduction is done. Naturally, the problem has an extremely high dimensionality, as each unique word is a dimension. However, words that appear very rarely, such as individual email addresses, provide very little information. Additionally, words that appear very frequently such as "a," "and," "the," etc. also provide very little information. The most common and least common words are therefore ignored, such that the number of dimensions M is made to respect the constraint $M < \frac{N}{2}$. Finally, the matrix is fed to a trained Support Vector Machine [10] classifier. For each revision i it returns $bugFix_i$, which is a real number between 0 and 1, indicating the confidence that revision i is a bug fix.

The final phase generates the ranked list of revisions. For each revision $1 \leq l \leq N$, a weight w_l is computed according to the following equation:

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

where $D(i,j)$ is the Euclidean distance from the line of HDL source s_i^j to its cluster exemplar and R_l is the set of lines changed by revision l. It can be seen that revisions containing lines closer to the cluster exemplar have a lower weighting. Additionally, revisions believed to not be bug fixes have a lower weight. As such, revisions with lower weight are ranked higher.

The ranked list is computed by first computing one ranked list per cluster. The list for a cluster contains only revisions that match HDL code in that cluster. These revisions are simply sorted in ascending order of weight. Subsequently, the lists are merged into a single master list. This process is explained in the following example, where lists A and B are the lists for each of the two clusters, and list C is the merged list.

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

The list is intended to be used as follows. The engineer should first examine revision R_1, as it has been identified as highly suspect and on its own explains both failures. If it is found not be the root cause, revisions R_2 and R_3 should then be examined, as they are the next most suspicious and together they can explain both failures. The engineer should proceed through the list in this fashion until a suitable fix is implemented.

III. Non-linear Revision Debug

This section describes the novel methodology. A high-level overview is presented first, in order to give an intuitive understanding of the flow. The algorithm is then presented in more detail.

The linear revision debugging algorithm of [4] does not make use of branch information. In modern version control systems, branches are commonly used to isolate development of a single feature or bug fix. One or more revisions are associated with each such branch. This information can be leveraged to improve classification performance. By considering branches in addition to revisions, a ranked branch list can be generated.

An overview of the proposed flow is shown in Fig. 5. The failure clusters (Fig. 5(a)) and revision bug fix probabilities (Fig. 5(b)) are computed similarly to those in [4]. Fig. 5(c) introduces the novel addition that makes use of branch information. Branch information is passed to an SVM classifier to determine the probability that each branch is a bug fix. This step is shown in more detail in Section III-C. Branches, and revisions associated with each branch, are then ranked, as seen in Fig. 5(d), and detailed in Section III-D.

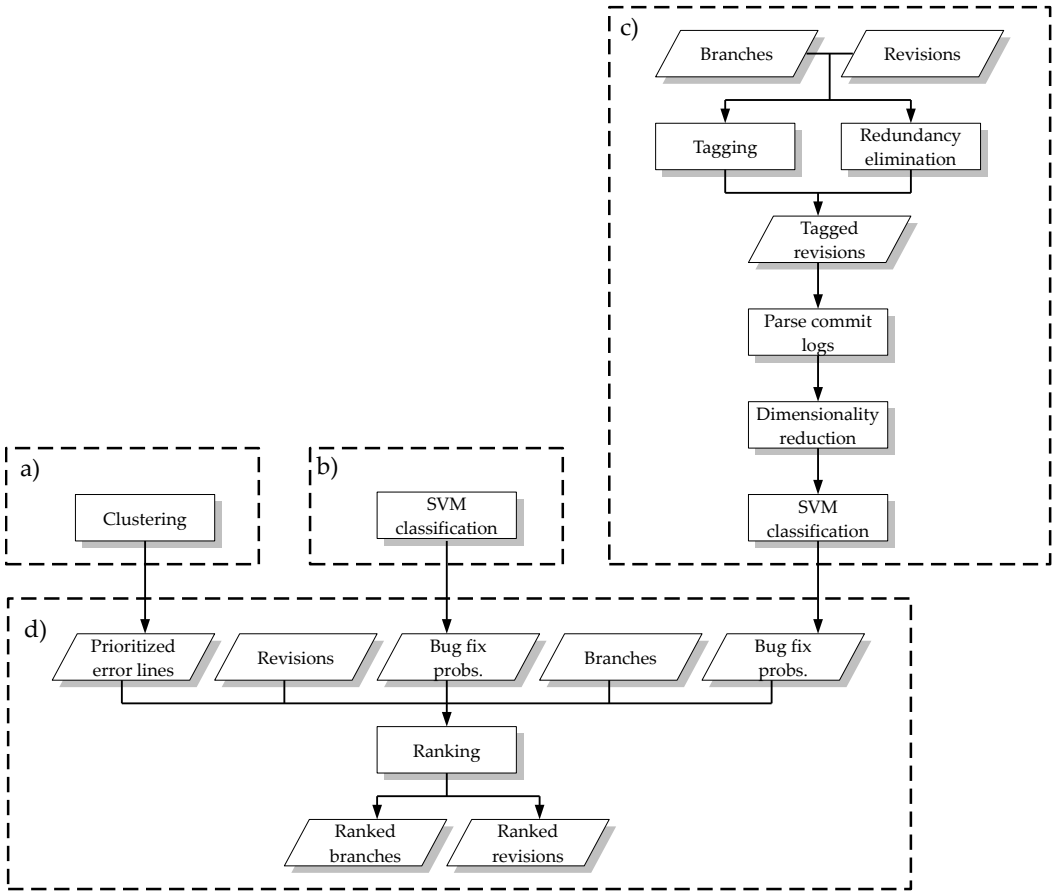

Fig. 5. Non-linear revision debug flow

A. Motivations

The intuitive motivation behind leveraging branch information is twofold. First, branches contain a natural grouping of similar revisions. Usually, if a branch is made for a bug fix, all revisions associated with the branch would be considered bug fixes, and vice versa. Revisions that would be difficult to classify when isolated can be assigned a more accurate class when considered as part of a whole.

Second, many revisions within a branch may be *redundant*, i.e. will not affect the mainline that experienced the failure. In the case of a feature addition, it is common to insert debugging code or comments during development. Some or all of this code will be removed before the branch is merged back into the mainline. These additional revisions do not contribute any useful information, and can be ignored. By considering branches as a whole, the effects of these revisions will naturally be eliminated with no additional work.

B. Branch Analysis

By modelling the revision history as a DAG, as seen in Fig. 1, branch nesting is determined and redundant branches are identified.

Using a depth-first search (DFS) on the DAG representation, revisions along the mainline are visited first. Revisions branching off the mainline are visited next, and so on. As revisions are visited during the search, they are tagged with the appropriate branch nesting level.

During the same graph search, redundant branches can be identified, for example the branch consisting of commits 6 and 7 in Fig. 1. If the DFS visits the last revision of a branch that is not merged (either into the mainline or into a branch that is eventually merged into the mainline), then it is a redundant branch. Revisions tagged as belonging to a redundant branch are considered redundant revisions, and can be ignored for classification and ranking, as they do not contribute to the mainline that encountered a failure.

C. Branch Classification

In order to classify branches, revisions are first tagged with branches they belong to. In the case of nested branches, revisions are tagged with more than one branch. More formally, let $B = B_1, B_2, ..., B_{|B|}$ denote the branches. Each revision R_l is tagged with a subset of the branches $B^l \subseteq B$. A combined commit log for each branch is generated by concatenating the commit log of each revision tagged with the branch.

In order to improve classification performance, functionally redundant revisions (i.e. revisions that do not affect the final functionality of the branch) will be excluded. To this end, a combined diff for each branch is generated by accumulating the diff of each revision tagged with the branch. These combined diffs are used to locate any redundant lines that do not affect the final contribution of the branch. Redundant lines are lines in the diff of a revision that are completely negated by lines in the diff of another revision within the same branch. Redundant lines for each revision are tagged as such, as this information will be used later during the ranking process.

Once all the redundant lines are determined, revisions that only contain redundant lines are redundant revisions. The commit logs for such revisions are then excluded from the combined commit log for

each branch.

Now that each combined commit log only contains useful commit logs, they are passed to an SVM classifier, similarly to [4]. In order to ensure that the classifier performs adequately with combined commit logs, it is trained using branches labelled as bug fixes or not, rather than single revisions. The classifier outputs a probability that each branch B_b is a bug fix, $bugFix^b$. Each revision tagged with the branch is assigned a probability of being a bug fix. For nested branches, the probabilities of the innermost branch take precedence (for example, a bug fix branch within a large feature branch would be considered a bug fix).

D. Weighted Branch Ranking

Branch ranking is performed similarly to revision ranking in [4]. Candidate lines that match changes made by non-redundant revisions are mapped to branches: $s_i^j \in B_b$. The weight w^b of each branch is then calculated with the following formula:

$$w^b = min_{i,j}\Big(\frac{1}{2}\Big(\frac{D(i,j)}{max_{i,j}(D(i,j))} + bugFix^b\Big)\Big) \quad \forall i,j | s_i^j \in B_b \tag{2}$$

Intuitively, the weight of a branch is the minimum weight of all the non-redundant revisions tagged with the branch. The higher the probability that the branch is a bug fix (i.e. not a feature addition), the higher the weight. Because of this, branches with lower weight will be assigned a higher rank.

Once weights for each branch are calculated, lists of locally ranked branches are generated. Each list corresponds to a single cluster from Section III-C, and contains the branches whose non-redundant revisions match candidate lines assigned to the cluster. The branches in these lists are sorted in ascending weight order.

To generate global branch rankings, the lists are merged. The branches in the merged list are the union of the branches at the same index in the individual lists. An example is shown below:

$$A = \begin{pmatrix} B_1 \\ B_2 \\ B_4 \\ \cdots \end{pmatrix}, B = \begin{pmatrix} B_1 \\ B_3 \\ B_4 \\ \cdots \end{pmatrix}, C = \begin{pmatrix} B_1 \\ B_2, B_3 \\ B_4 \\ \cdots \end{pmatrix}$$

The merged list C contains the final ranked branches. The highest ranked branch is B_1, as it is ranked highest in each of the individual lists A and B. The second highest ranked branches are B_2 and B_3, with the same ranking, as the are both ranked second in different lists.

Finally, revisions within each branch are locally ranked. Using a similar ranking scheme as above, merged lists for branch-local non-redundant revisions are generated. For each branch, lists are generated for each cluster containing only revisions tagged with the branch, which are then sorted in ascending weight order. These lists are then merged to get the branch-local ranking of each revision.

IV. Experimental Results

This section presents experiments results. The linear revision debugging methodology of [4] is contrasted against the proposed non-linear methodology. All experiments are conducted on a workstation with an Intel Core i5-3570K processor clocked at 3.40 GHz, with 16 GB of RAM. A total of nine RTL designs are used – eight from `OpenCores` [11] and one in-house design, with Subversion revision history available for each design. From these designs, a total of 18 testcases are generated.

A SAT-based automated debugging tool based on [2] is used to find candidate lines. A Python platform is used to parse revision and branch information, along with performing AP clustering, classification, and ranking. Branch classification is performed without a priori knowledge of whether branches are bug fixes or not. Prior to running experiments, an SVM classifier is trained for this task. Finally, branches and branch-local revisions are ranked for each testcase.

Individual testcases are generated from golden design by injecting errors into the RTL. Injected errors are generated by reverting previous bug fixes that were made in the revision history. Anywhere between one and three different subsets of these errors are selected for each design, then injected to create each of the 18 testcases. The testcases are run against the pre-existing testbench and erroneous responses are recorded, which are passed to the SAT-based automated debugging tool.

The SVM classifier is trained using a separate script to obtain a prediction model. To this end, revisions are first parsed for branch information, then tagged with branches. Dimensionality reduction is subsequently performed on the commit logs in order to remove unnecessary words and mitigate the "Curse of Dimensionality." Remaining words after dimensionality reduction are used for both training the SVM and branch classification.

Design and testcase information is summarized in Table I. The columns in this table are arranged as follows. The first three columns show design name and testcase number, followed by the number of logic elements in the synthesized design. The next two columns show the number of errors injected into testcase, and the resulting number of failures. The last two columns show the total number of revisions and branches in the repository. It is of interest that the number of branches is, for most cases, substantially lower than the number of revisions.

TABLE I
TESTCASE STATISTICS

Design	Test Num.	Logic Elem.	Num. Err.	Num. Fail.	Num. Rev.	Num. Br.
ethernet	1	76408	6	10	332	37
	2	76408	1	4	332	37
HA1588	3	9152	4	6	70	6
	4	9152	3	6	70	6
	5	9152	7	12	70	6
I2C Core	6	3640	3	4	70	2
	7	3640	3	12	70	2
Tate Pairing	8	106786	4	4	33	5
	9	106786	5	37	33	5
SD Card	10	38211	1	20	127	43
	11	38211	4	36	127	43
SDR Ctrl	12	18374	2	5	72	22
	13	18374	8	10	72	22
6507 CPU	14	9416	2	3	259	51
	15	9416	2	3	259	51
VGA	16	109797	3	5	59	8
Packet Forwarder	17	40197	2	16	177	4
	18	40197	4	23	177	4

Table II compares the performance of the linear revision debugging technique [4] with our proposed methodology. Ranks are of the branch/revision(s) responsible for the failure. The first column in the table shows the testcase index. The next three columns show the performance of [4], the first two showing the rank achieved and the minimum rank, while the last shows the runtime. The next five columns are arranged similarly, with the first two showing branch ranking and minimum, the next two showing branch-local revision ranking and minimum, and the last showing the runtime. Finally, the last column shows the improvement of branch ranking over revision ranking of [4]. This is calculated as $(1 - (Br.Rank/Rev.Rank) *$

TABLE II
REVISION RANKING PERFORMANCE

Test Num.	Linear [4]			Non-Linear					improv. (%)
	Rank		Time	Rank				Time	
	rev. rank	rev. total	(s)	br. rank	br. total	local rank	local total	(s)	
1	6	27	4.335	2	11	2	6	6.716	67
2	5	10	6.389	1	9	4	8	9.830	80
3	1	5	0.878	1	3	1	4	2.163	0
4	1	7	3.322	1	4	1	6	5.334	0
5	1&2	7	3.432	1	3	1&2	3&3	5.792	0*
6	1	13	2.147	1	2	1	12	4.471	0
7	1&3	14	1.931	1&2	2	1&1	5&4	3.560	33
8	4	6	0.710	2	4	2	5	2.043	50
9	1&2	3	0.611	1&2	3	1&1	5&7	1.284	0
10	4	9	3.213	2	6	3	4	7.672	50
11	2	10	3.022	3	9	1	6	8.021	-50
12	2	12	27.328	1	8	2	7	40.631	50
13	2	13	17.976	1	5	2	3	35.557	60
14	35	49	2.339	4	14	7	11	4.169	89
15	41	48	2.486	6	17	5	8	4.428	85
16	11	15	1.126	3	7	3	5	3.830	73
17	2	15	0.905	1	3	2	3	2.095	50
18	4&8	16	1.207	2	4	2&6	4&7	3.436	50*
AVG.	7	16	4.630	2	6	3	7	8.391	38

100%. In the cases where a single branch with two revisions was the root cause, the higher ranked revision is used.

Branch ranking invariably takes longer than revision debugging, as both branches and revisions must be ranked. This is seen as a 81% average increase in runtime. However, both ranking systems take only a fraction of the total time spent generating candidate lines using automated debugging, which takes anywhere between 400 and 3600 seconds.

The effectiveness of branch ranking can be seen in tests `1-2`, `10`, and `12-15`. For these cases, the responsible branch is ranked higher than the responsible revision from linear revision ranking. The end result is that engineers would find the root cause of the failure sooner, as they can begin looking at the responsible set of changes sooner. Another factor that makes branch ranking useful is exemplified in tests `5` and `18`. In these cases, all the responsible revisions are located in a single branch, so the engineer would only need to look at one branch in isolation to determine the root cause, rather than look through two or more potentially distant commits.

Branch ranking provides less useful results in cases where the number of branches is too small, such as in tests `6-7`. In these cases, branch ranking does not outperform revision ranking by any meaningful margin. Finally, it is possible for branch ranking to rank the responsible branch lower than the responsible revision of revision ranking. This is seen in test `11`. This can occur because branches cover more changes than single revisions. A branch that matches with more suspects could end up ranked higher than the responsible branch, if the latter matches with fewer suspects.

In order for branch ranking to perform optimally, good coding practices should be followed, were branches are used for any feature additions or bug fixes. Changes made in branches should also be isolated to a single feature or bug fix.

V. CONCLUSION

This paper introduces a novel extension for revision debugging that makes use of branch information available in revision history. The methodology identifies redundant revisions and lines, then ranks branches and branch-local revisions. An extensive set of experiments demonstrate its benefits over a linear revision debugging approach, with improved ranking performance and a marginal runtime increase. A promising direction for future work in this area includes leveraging features available in issue tracking systems to enhance basic revision history.

REFERENCES

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

[2] 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.

[3] R. Berryhill and A. Veneris, "A complete approach to unreachable state diagnosability via property directed reachability," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2016, pp. 127–132.

[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, pp. 32–37.

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

[6] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections: Hit or miss?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 322–331. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025157

[7] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 296–305. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081754

[8] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 279–289.

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