Leveraging Software Configuration Management in Automated RTL Design Debug

John Adler and Andreas Veneris
University of Toronto

Editor's note:
This article presents an enhancement to the existing automated debugging software by leveraging statistics from the revision control history.
—Li-C Wang, University of California at Santa Barbara

Functional verification of hardware designs is the major bottleneck in the design cycle today, accounting for up to 70% of the total time [1]. The majority of this effort is taken by debugging, a process that identifies errors once verification demonstrates a failure. As debug remains a semiautomated task, new tools and methodologies are needed to contain the pain and improve the modern verification cycle.

Traditionally, given a set of counterexamples where the design fails, debug returns suspect lines that may contain the error source(s). In the modern verification cycle, these counterexamples are generated by simulation and formal tools that exercise different parts of the design's functional behavior. Failures are usually recorded by monitors, checkers, scoreboards, or assertions. When a failure is exposed, the counterexample(s) are used by the engineer to manually trace through the design with the help of waveform viewers to discover the error, or by debugging tools that automatically return suspect locations. These debug tools predominantly use a combination of simulation and formal techniques, such as path-trace, Boolean satisfiability (SAT) and binary decision diagrams, to prune the solution space and look for suspects [2]–[4].

Waveform-based debug tools allow users to backtrace through the circuit using information obtained from the counterexamples. They can also perform what-if analysis and resimulation on the fly, allowing users to inspect how changes to the circuit will propagate forward. This aids the debug process, but still requires a significant amount of manual effort to decide where changes should be made. Simulation-based automated debug tools researched in the 90s [2] perform this what-if analysis exhaustively, simplifying decision making in root cause error analysis. BDD-based [4] automated debug, on the other

Digital Object Identifier 10.1109/MDAT.2017.2713391
Date of publication: 8 June 2017; date of current version: 13 September 2017.

hand, is accomplished by modeling the failing circuit as a binary decision diagram. These techniques make tradeoffs between resolution and memory utilization which may limit their applicability.

More recent research in SAT-based debug [3] provides a scalable and extensible platform when compared with previous techniques. The original design is enhanced with hardware and then modeled as a SAT problem which is constrained by simulation input–output values and passed to a SAT solver. Given the specific constraints, the SAT solver will attempt to find a set of locations that must be changed to correct the circuit. Based on the original SAT-based debug formulation, various enhancements have been proposed to improve runtime and memory requirements, most notably abstraction and refinement in the space and time domains [5].

Evidently, research in debug automation for the past 20 years has been design-centric. In other words, it focuses on an analysis of the design functionality over a finite set of input stimuli to detect and locate the root cause of the failure. An important aspect of the design cycle that has been overlooked during debug is the wealth of information during its development history. This is usually contained in version control systems (VCSs) and issue tracking systems (ITSs), collectively known as software configuration management (SCM). These systems have become a necessity in today's geographically decentralized design world that integrates new components with legacy and the third-party intellectual property ones to build circuits comprised of hundreds of millions of gates. In other words, the human insight and past engineering effort recorded in historical revisions and associated metadata is largely ignored by existing automated debug methodologies.

This paper describes recent developments in debug automation that factors in this significant information with the use of statistical techniques to expedite the task. The novel methodologies proposed here complement existing design-centric debug tools with results from human-centric historical data that is automatically parsed by machine learning algorithms. Empirical results presented here confirm that when traditional debug is collectively enhanced with practical knowledge contained in SCM, the overall task is simplified in the continuous effort to further alleviate the burden behind the verification/debug task.

```
1  Commit <f1d2d2f924e986ac86fdf7 36c94bcdf32beec15>
2  Author: John Doe <john.doe@domain.com>
3  Date: Fri Jan 1 12:34:00 2016 -0000
4
5  refs #128
6
7     Fixed incorrect bus range.
8
9  --- a/top.v
10 +++ b/top.v
11
12 - input [5:1] b;
13 + input [4:0] b;
```

Figure 1. Sample revision metadata.

Software Configuration Management

The two facets of SCM that contain both pertinent and easy-to-parse information are VCSs and ITSs. The former provides a record of all changes made to a design in the form of revisions, while the latter enhances revision metadata.

VCSs, such as Git or Subversion, are widely used as they enable multiple designers from potentially distant locations to work on a single project in tandem with ease. Each time a change is made to the design, a revision records the change and associated metadata. As exampled in Figure 1, a sample revision metadata usually contains a unique revision id (line 1), the time the change was committed (line 3), the user that made the commit (line 2), and a user-specified message describing the change (lines 5–7). Changes are usually in `diff` form, which show differing lines before and after the change is made, for each file changed (lines 9–13).

In their simplest form, revisions are ordered as a linear list, i.e., the changes for each revision are applied in succession. Modern VCSs also support branching schemes, allowing for isolation of the development of a single feature or bugfix, as shown in Figure 2. This branching structure can be explored as a Directed Acyclic Graph (DAG). The mainline, or master branch, here comprised of revisions R0, R1, R2, and R7, is analogous to the linear list of revisions, with merge revisions (R7) applying the cumulative `diff` of a branch onto the mainline.

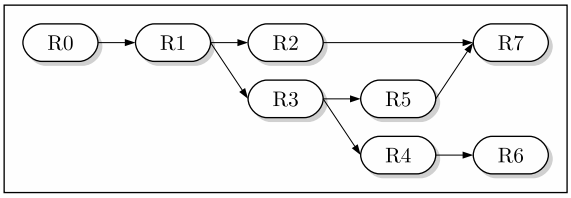

Figure 2. Revision history as directed acyclic graph.

ITSs, which come in a much wider variety of flavors than widely adopted VCSs, allow for additional complementary information to be recorded about revisions and their relationships. Issues can be associated with revisions and branches. For example, the revision in Figure 1 is linked to issue #128 (line 5). Likewise, issues can be tagged as being bugfixes or features. Additional relationships between branches and/or revisions can also be represented: for example, one branch might fix a bug introduced in the previous branch. Different ITSs can provide numerous other features, but the aforementioned features are the most directly relevant to the debug problem described here.

Statistical Automated Debug

Revision debug ranks revisions based on their probability of having introduced an error into the design. Formal- and simulation-based debug techniques provide no ranking information for the suspects to the engineer. Being able to rank revisions allows the engineer a starting point to prioritize analysis and provides a starting context to the process, ultimately reducing the expected number of suspects that must be examined and speeding up the debug process. This is accomplished by enhancing results from a formal debug tool using information available in the SCM system. A variety of techniques using machine learning have been proposed to give meaning out of this human-centric information. Clustering-based revision debug [6] accomplishes this by matching revision changes with approximate error sources. Perceptron-based revision debug [7] ranks revisions using a trained classifier.

The key motivator behind these techniques is making use of the previously untouched human-usable information in the form of revisions and issues. Engineers separate their coding into commits, write detailed commit messages, and manually create and close issues, all in an effort to provide context to their work to other engineers. This information can be used by properly tuned machine learning techniques, a task which is impossible for traditional formal techniques.

Clustering

Performing revision debug using clustering can provide revision ranking immediately without the need of a large training set, making it more effective for designs with a short history. This technique is separated into three distinct steps: suspect clustering, revision classification, and weighted revision ranking. It can also optionally be extended to rank branches in addition to revisions [8].

Affinity propagation clustering [9] is a clustering algorithm that can automatically partition into groups a set of data points. Contrary to some simpler clustering algorithms, it can determine the number and location of exemplars (cluster centers) without user input. This is important because clustering is used to estimate the number of errors in the design that caused the observed failures, which is unknown and cannot be reliably estimated by the user.

Given a set of failures $F = \{f_1, \ldots, f_{|F|}\}$, for each failure f_i, a formal debug tool returns suspects $S_i = \{s_1^i, \ldots, s_{|S_i|}^i\}$. Each suspect is a set of RTL lines that, if changed appropriately, will fix the erroneous behavior. Suspect clustering is used to group these suspects. Each suspect s_j^i is mapped to a space based on its location (i.e., file and line number). This is illustrated in Figure 3, where the suspect sets of five failures across two Verilog files are mapped to a two-dimensional space. The axes correspond to line numbers for each file. Affinity propagation clustering [9] is then used on these mapped suspects to automatically locate exemplars. Each exemplar represents an error source, therefore, the number of exemplars corresponds to an estimation of the

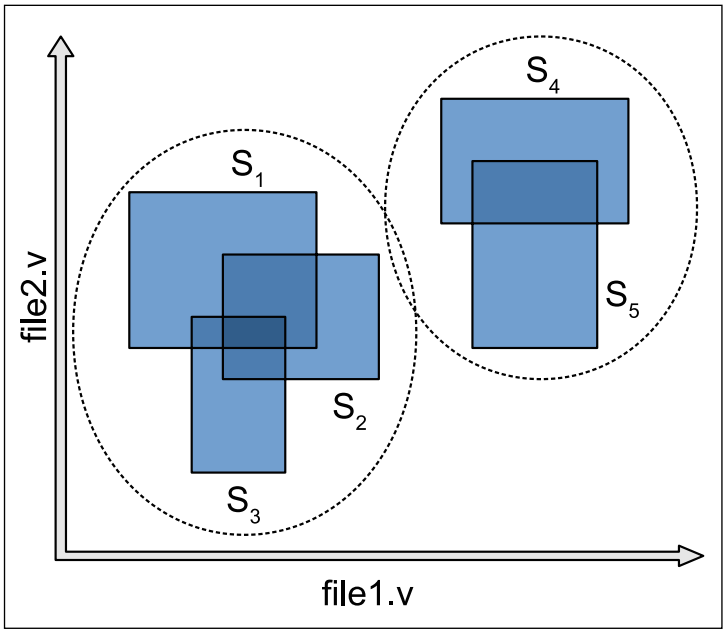

Figure 3. Clustering-based revision debug: clustering step.

number of errors in the design. The Euclidean distance D_j^i between each suspect s_j^i and its exemplar is calculated, as it will be used during weighted revision ranking. Intuitively, this distance corresponds to the proximity between a suspect and an error source.

Next, revision classification is performed, with the goal of tagging each revision as either a bugfix or not. If the SCM includes an issue tracker, where revisions are manually pretagged as such, then this step can be replaced by using the ITS information directly. To perform classification, a support vector machine (SVM) [10] is trained on labeled commit messages. This type of classifier is used since it can predict the probability that an input sample belongs to a certain class. Once trained, the SVM classifier is used to predict the probability P_k that the kth revision R_k is a bugfix or not. Intuitively, revisions that are bugfixes, are less likely to have introduced an error into the design, which in effect can be factored into the weighting.

Finally, weighted revision ranking is performed using results from the previous steps. Suspects are mapped to revisions by comparing suspect locations to revision `diff`s. For each revision R_k, a weight is assigned

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

The minimum sum between the Euclidean distance and the bugfix probability is used as the weight. Intuitively, the lower the assigned weight, the more likely the revision has introduced an error. Revisions that have suspects closer to an exemplar, i.e., made changes in close proximity to an error, will have a smaller weight. Revisions that are bugfixes, are less likely to introduce errors, and so will have a large probability P_k, increasing the weight.

This weight can now be used to sort revisions and determine a relative ranking. For each cluster C_i, the list of revisions with suspects in that cluster is sorted by ascending weight. The lists are then merged, with revisions in the same position being equally likely to have introduced an error. For example, the list of revisions for two clusters C_1 and C_2 are merged into unified list C':

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

In cases where a revision has multiple final rankings, the highest one is taken.

Perceptron

Perceptron-based revision debug takes a different approach, using a trained perceptron to automatically determine how to combine SCM data with results from formal debug techniques. Perceptrons [11] are another machine learning technique that can be used as classifiers. Generally, perceptrons are trained on labeled data, and then the trained perceptron can be used to predict the class of new samples. This has the advantage over clustering of being able to incorporate more of the information available, but requires an extensive amount of training samples to outperform the aforementioned approach. Perceptrons are akin to single-layered neural networks, and can be implemented using techniques such as Logistic Regression and SVMs [10].

The first step is flattening the revision history. Since perceptrons are trained on a list of samples, the branching structure of revisions must be transformed into a linear list with minimal information loss. To this end, two alternatives are available: revision-to-revision or revision-to-head. The former uses the changes of each revision directly, which is easier to implement but can cause information loss where branches are merged. The latter option generates changes by taking the `diff` between each revision and the head, the latest revision on the mainline (or, the revision where the design failure is observed). To do this, the revision history is traversed using a depth-first search (DFS) starting at the head. Whenever a merge is encountered, the DFS visits the branch first before returning to traverse the parent branch. For each revision visited, the `diff` between this revision and the head is calculated. This is then compared to the `diff` of previous revision to remove redundant changes. Intuitively, revision-to-head `diff`s represent the effects of a revision on the head, rather than on the previous version of the design.

Once the revision history is flattened with either option, training samples must be generated. The previous failures and their fixes will be used to train the perceptron. To start, a set of failing revisions throughout the design's history is selected, designated as *secondary heads*. Each such head will serve as a base from which to generate revision-to-head `diff`s and

label previous revisions. It is required to run revision-to-head flattening for each secondary head since the generated `diff`s represent how a revision specifically affects the head used during flattening. For each secondary head, a set of previous revisions is selected and labeled as not having inserted an error into the design. In addition, the erroneous revision for that failure is extracted from the ITS and labeled as having inserted an error.

In order to complete the generation of training samples, suspects must be incorporated. Since a perceptron requires a fixed number of features as input, suspect information must be encoded as such. For each revision R_k, the matching value V_l^k represents the number of suspects matching changed line l in the revision. Exact matching is not required, and an exponentially decaying weighted distance can be used instead

$$V_l^k = \sum_{s_j^i \in S_i} e^{-f_e \cdot Dist(l, s_j^i)} \quad \forall 1 \leq i \leq |F| \tag{3}$$

where the distance *Dist* is the absolute difference between changed line l and suspect s_j^i and f_e is an experimentally tuned matching constant. Intuitively, the farther a changed line is from a suspect, the less they "match" and so the smaller the matching value is. The set of matching values for each revision can then be trivially encoded as a fixed-length list. Now that a set of training samples has been generated, the perceptron can be trained. A graphical illustration of input features to the perceptron is shown in Figure 4. The revision and branch ID, along with whether the revision is a bugfix or not, is concatenated to the list of matching values. The two IDs will provide differentiating power in cases where a single revision or branch has inserted multiple errors into the design that manifest as several failures throughout the design's history. When training, the output of the perceptron is fixed with the labeled data.

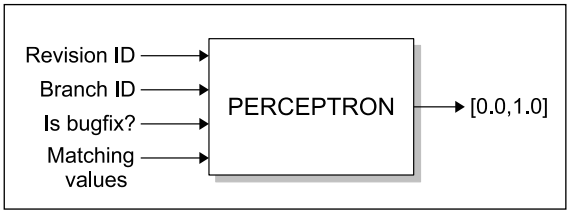

Figure 4. Input/output of the perceptron.

Once the trained perceptron is ready, it can be used to predict the probability that a given future revision has introduced an error, or locate a past revision that has inserted an error but remained undetected. This has an advantage over the clustering-based method in that this probability is an absolute one, rather than the relative revision ranking produced by clustering.

Experimental Results

Three different approaches are compared here: a manual, brute-force approach, described in [6], the clustering-based approach [6], and the perceptron-based approach [7]. The brute-force approach involves manually backtracing through the design until the root cause is located, matching revision changes to explored RTL locations. Each test is conducted on a workstation with an Intel Core i5-3570K CPU running at 3.40 GHZ with 16 GB of memory. Revisions and issues are gathered from each design's SCM system. A SAT-based automated debug tool based on [3] is used to generate the suspects needed for both revision debug methods. The above data is parsed and combined with Python scripts. To generate each test case, a *target revision* is selected. This is a revision that had previously introduced an error into the design. The correction to the error is rolled back to create a failing design. The goal of each revision debug approach is the rank this target revision (i.e., test) as highly as possible.

Table 1 summarizes pertinent design information and provides a comparison between the three approaches. For each design, the second column shows the number of logic elements, the next two contain the number of heads and number of features for the perceptron-based approach, and the fifth column contains the number of revision in the design's VCS. The number of features is directly proportional to the number of matching values input into the perceptron, while the number of training samples is directly proportional to the number of heads. The sixth and seventh columns show the determined ranking of the target revision and runtime for the brute-force approach. The next two columns show the same for the clustering approach. The last three columns show the rank of the target revision, the output value of the perceptron (which can be interpreted as the confidence that the target revision introduced an error), and the runtime for the perceptron-based approach. Evidently, both automated

Table 1 Revision ranking performance.

Design	Logic Elem.	Num. Heads	Num. Feat.	Num. Rev.	Brute force [6]		Clustering [6]		Perceptron [7]		
					rank	time (s)	rank	time (s)	rank	y	time (s)
6507 CPU	9416	42	240	259	27	0.180	41	2.408	5	0.79	0.598
ethernet	76408	87	992	368	58	0.257	6	4.726	4	0.87	1.109
HA1588	9152	26	236	70	11	0.294	1	1.091	7	0.88	0.822
I2C Core	3640	30	673	76	23	0.771	1	2.165	3	0.83	0.972
pkt. fwd.	40197	91	88	177	15	0.928	8	1.153	13	0.74	0.503
SD card	38211	54	541	137	19	1.365	4	3.217	9	0.72	0.806
SDRAM ctrl	18374	13	2109	72	19	1.532	2	25.309	25	0.51	1.528
tate pairing	106786	9	227	33	16	0.109	4	0.592	8	0.60	0.682
VGA	109797	15	303	64	3	0.293	12	1.384	16	0.66	0.699

techniques provide better rankings than the manual approach, with minimal runtime overhead, demonstrating their practicality.

While the perceptron-based approach performs, on average, poorer than the cluster-based approach, its potential is demonstrated in Figure 5. The learning curves for the `ethernet` test case using twofold cross validation are shown, and it can be seen that they have not converged within the training samples used. This means the perceptron-based approach has the potential to give more accurate predictions if additional training data is available. In the cases of large industrial designs with long histories, this data is readily available. In addition, for industrial workflows, it is trivial to use each new bugfix as a new head to train with, allowing the perceptron-based approach to integrate seamlessly.

More than half of the modern verification cycle is spent on debugging. Existing automated debug techniques are design-centric in the sense that they examine the design and the failed vectors from verification. This paper presents novel methodologies to improve the automated debug process using the human-centric information of a design's history. In those techniques, results from existing formal debug techniques are combined with SCM data using two different approaches: clustering and perceptrons. The net result is a precise ranking of the design's revisions to help the engineer discover the root cause of failure faster.

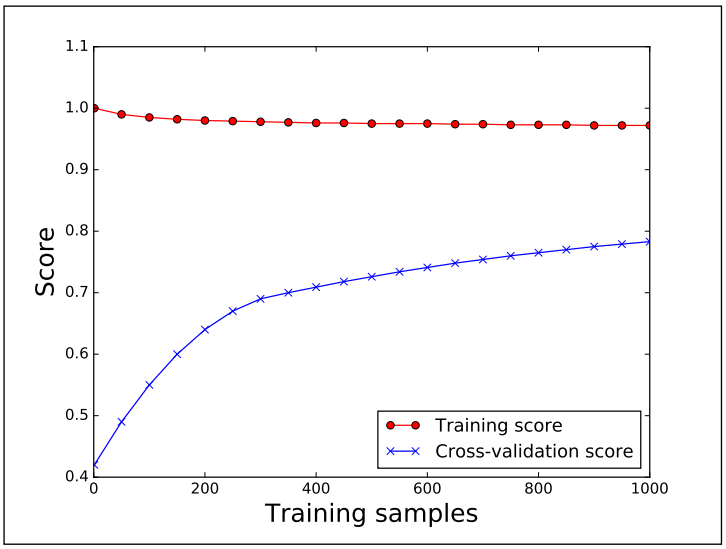

Figure 5. Learning curves for ethernet perceptron.

THIS AREA OF RESEARCH is new and promising. For example, the perceptron-based approach can be extended to include a variety of additional information available in the VCS and ITS, including commit time, commit message, individual committing, branches, commit size, etc. Further, other machine learning techniques can also potentially be utilized to generate revision rankings and further improve the design verification cycle. ■

References

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

[2] S.-Y. Huang et al., "Errortracer: A fault simulation-based approach to design error diagnosis," in *Proc. Int. Test Conf., 1997*, Nov. 1997, pp. 974–981.

[3] A. Smith et al., "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] P.-Y. Chung, Y.-M. Wang, and I. N. Hajj, "Logic design error diagnosis and correction," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 3, pp. 320–332, Sept. 1994.

[5] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 11, pp. 1790–1803, Nov. 2010.

[6] D. Maksimovic, A. Veneris, and Z. Poulos, "Clustering-based revision debug in regression verification," in *Proc. 33rd IEEE Int. Conf. Comput. Design (ICCD), 2015*, Oct. 2015.

[7] J. Adler, R. Berryhill, and A. Veneris, "An extensible perceptron framework for revision rtl debug automation," in *Proc. 22nd Asia and South Pacific Design Autom. Conf.*, 2017.

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

[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] E. Alpaydin, *Multilayer Perceptrons*. MIT Press, 2014, p. 640.

John Adler is currently an MASc student at the University of Toronto, Toronto, ON, Canada in Electrical and Computer Engineering. His research is on CAD for design debug and verification using formal and statistical methods. He has a BASc. in engineering science from the University of Toronto. Contact him at adler@eecg.toronto.edu.

Andreas Veneris is a Professor of Electrical and Computer Engineering and Computer Science at the University of Toronto, Toronto, ON, Canada. His research is in CAD for debug, verification, synthesis, and test of digital VLSI circuits and systems. He has a PhD from the University of Illinois at Urbana-Champaign, Champaign, IL, USA. He is a senior member of IEEE. Contact him at veneris@eecg.toronto.edu.

■ Direct questions and comments about this article to John Adler, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada; adler@eecg.toronto.edu.