VeryBug: An Attention-based Framework for Bug Localization in Hardware Designs

VeriBug: An Attention-based Framework for Bug Localization in Hardware Designs

Giuseppe Stracquadanio[1,2], Sourav Medya[1], Stefano Quer[2] and Debjit Pal[1]
[1]*University of Illinois Chicago, Chicago, IL, USA* [2]*Politecnico di Torino, Torino, Italy*
{gstrac3, medya, dpal2}@uic.edu, stefano.quer@polito.it

***Abstract*—In recent years, there has been an exponential growth in the size and complexity of System-on-Chip (SoC) designs targeting different specialized applications. The cost of an undetected bug in these systems is much higher than in traditional processors, as it may imply loss of property or life. Despite decades of research on simulation and formal methods for debugging and verification, the problem is exacerbated by the ever-shrinking time-to-market and ever-increasing demand to churn out billions of devices. In this work, we propose VeriBug, which leverages recent advances in deep learning (DL) to accelerate debugging at the Register-Transfer level (RTL) and generates explanations of likely root causes. Our experiments show that VeriBug can achieve an average bug localization coverage of 82.5% on open-source designs and a wide variety of injected bugs.**

I. Introduction

Simulation and formal verification are two complementary verification techniques. Given a design property, formal verification proves the property holds for every point of the search space. Simulation, instead, verifies the property by pseudo-randomly testing a small subset of the search space.

Bug localization techniques relying on formal methods, such as BDDs, BMC, Interpolants, IC3, and other SAT-based methods offer a systematic and rigorous approach to identifying and localizing bugs in hardware designs. However, they are *computationally intensive*, require *additional expertise*, and require *conspicuous effort for specifying complex mathematical logic models*. Bug localization in simulation-based workflows is addressed by identifying common patterns [5] in failure traces and mapping them to the source code. Unfortunately, prior localization techniques often lack explanation for a potential bug, leaving the reasoning to the verification engineer.

These limitations, coupled with the ever-increasing hardware complexity and shrinking time-to-market, mandate the development of new hardware verification methods.

Following this research direction, we propose VeriBug, an automated bug-localization framework that harnesses the power of a data-driven approach and recent advances in deep learning (DL). The major drawbacks of other DL-based verification techniques are as follow. First of all, they directly extract features from the specific code [6] and do not guarantee generalization to unseen code structures. Secondly, they mainly approach the problem as a *classification* task, where an entire program is classified as buggy or not buggy [3], thus requiring large annotated datasets to train neural networks. To overcome these constraints, VeriBug avoids dependence on features of a specific programming language and instead automatically learns relevant design features from abstract design representations such as Abstract Syntax Trees (ASTs). This unique capability to learn features from such representations makes VeriBug broadly applicable to hardware designs developed in other hardware programming languages. These learned features are *design agnostic*, allowing for generalization to unseen designs. Our DL architecture localizes the root cause of a bug by comparing the learned semantics for failure and the correct simulation traces. VeriBug does not need to be trained on a labeled design corpus as we train it on a *proxy* task to learn execution semantics from simulation traces. Moreover, the approach is fully integrable with current verification workflows without additional artifacts.

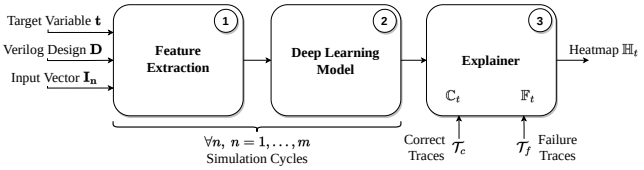

Fig. 1: **VeriBug workflow**. The Feature Extraction component (①) extracts features from the design. The Deep-Learning Model (②) learns the execution semantics. The Explainer (③) component aggregates trace-level semantics into condensed execution information and produce heatmaps.

Our experiments show that learned knowledge can be transferred and generalized to unseen designs. In our bug injection campaign, on four different real designs, we obtain an *average* coverage of 82.5%, and we localize 85 injected bugs over a total of 103 observable ones.

II. Overview of our VeriBug architecture

We aim to root-cause a design failure and localize it to a subset of the design source code. Given a Verilog hardware design $\mathbf{D}$, with $\mathbf{I}$ inputs, $\mathbf{O}$ outputs, and an observed failure f at output $\mathbf{t} \in \mathbf{O}$, VeriBug localizes the failure to a subset of likely suspicious source code statements $\mathcal{L} \subseteq \mathbf{D}$ and generates quantitative explanations of suspiciousness.

Our tool VeriBug introduces a novel bug-localization technique by leveraging state-of-the-art DL. Figure 1 shows the complete workflow of VeriBug, consisting of three components.

The first component, the *Feature Extraction* module, extracts knowledge (*i.e.*, raw features) from the input design $\mathbf{D}$. We use GoldMine [4] to extract a Control Data Flow Graph (CDFG) and a Variable Dependency Graph (VDG) of the Verilog design

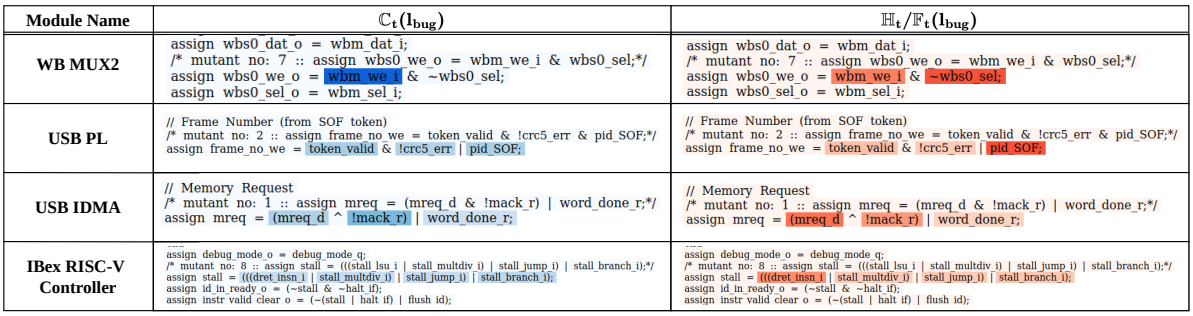

Fig. 2: VeriBug generated heatmaps on real designs. We report a visual comparison between operand importance scores in $\mathbb{C}_t$ (blue, deeper is more important) and importance scores in $\mathbb{F}_t$ (red, deeper is more important).

$\mathbf{D}$. We analyze the CDFG and the VDG to identify the control and data dependencies for $\mathbf{t}$ and to extract design slices, including relevant statements. All relevant statements are then translated into Abstract Syntax Trees (ASTs).

Our *Deep Learning Model* produces rich operand embeddings for each operand in a statement l_k, by encoding its input assignment and embedding AST paths that contain the operand. After that, it computes a weighted sum with operand embeddings using attention weights produced by our Attention [2] module. Given the AST of statement l_k and operands' input assignments, the model predicts its output value, together with attention weights that we interpret as *importance scores* for operands. Using this model architecture we enforce a strong inductive bias for learning execution semantics.

The *Explainer* module aggregates *importance scores* for correct simulation traces $\mathcal{T}_c$ and failure simulation traces $\mathcal{T}_f$, producing two different aggregated attention maps, respectively $\mathbb{C}_t$ and $\mathbb{F}_t$. The aggregation of such scores leads to a concise representation of the design execution behavior in the two simulation scenarios, allowing for a direct comparison for bug localization. Specifically, when the euclidean distance $d(\mathbb{F}_t(l_k), \mathbb{C}_t(l_k))$ is higher than a pre-defined threshold, we store $\mathbb{F}_t(l_k)$ scores in a heatmap $\mathbb{H}_t$. We also use $d(\mathbb{F}_t(l_k), \mathbb{C}_t(l_k))$ as a *suspiciousness score* for l_k. Following this heuristics, the final heatmap $\mathbb{H}_t$ contains only importance scores of candidate buggy statements.

III. Experimental Setup and Results

We train VeriBug on a synthetic data set to enforce variability in training data necessary to achieve generalization. During training, we create batches by sampling statements and their associated input vectors from the training set. Our attention-based neural network is trained to predict the output value of each statement. We assume that models with higher prediction accuracy can compute better statement representations and generally learn better features related to execution semantics. Thus, we select the model with the highest accuracy as the output of the first step in our evaluation pipeline. After the training phase, we evaluate the effectiveness of VeriBug for bug localization using realistic designs.

Notice that, at inference time, the last layers for prediction are discarded and only attention weights are used and aggregated to create $\mathbb{F}_t$ and $\mathbb{C}_t$. To validate the bug localization effectiveness, we inject bugs of varying complexity. Specifically, we introduce data-centric bugs by automatically mutating the designs according to a pre-defined set of mutation rules.

We compute a bug coverage metric for each design-target $\langle \mathbf{D}, \mathbf{t} \rangle$ pair as the ratio between the number of localized bugs and the total number of observable bugs. We consider a bug as localized when the highest *suspiciousness score* in the heatmap $\mathbb{H}_t$ is assigned to the statement containing the actual root cause. We observe that VeriBug effectively localizes different bug types in data-flow with a top 97.6% bug coverage, obtained on the IBex RISC-V Controller [1]. These experimental results demonstrate that VeriBug is a promising approach for precise bug localization of data-centric bugs. This empirical evidence also shows that the learned knowledge is transferable to real designs after synthetic training, making VeriBug a promising approach for broader adoption and scaling.

VeriBug allows for further localization via mapping importance scores stored in heatmap $\mathbb{H}_t$ back to the RTL code. We provide a few examples of VeriBug generated heatmaps on real designs in Figure 2.

IV. Conclusion

We presented VeriBug, a framework for bug localization which automatically generates explanations for bugs, to make decisions accessible to a debugging engineer. VeriBug is particularly effective for data-flow bug localization, and its learned knowledge is transferable to unseen designs. Albeit preliminary, our approach shows promising results, and it is compatible with current verification flows without any additional artifacts.

References

[1] IBex. https://github.com/lowRISC/ibex. Accessed: January 17, 2024.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *Int'l Conf. on Learning Representations (ICLR)*, 2016.

[3] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proc. of the ACM on Programming Languages (PACMPL)*, 2019.

[4] D. Pal, S. Offenberger, and S. Vasudevan. Assertion Ranking Using RTL Source Code Analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020.

[5] D. Pal and S. Vasudevan. Symptomatic Bug Localization for Functional Debug of Hardware Designs. *Int'l Conf. on VLSI Design Embedded Systems (VLSID)*, 2016.

[6] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: Bug detection with n-gram language models. *Int'l Conf. on Automated Software Engineering (ASE)*, 2016.