Using Machine Learning Techniques for Logic Design Verification

MUSTAFA GENCEL

Abstract – Logic Design Verification has been one of the most resource-demanding stages in digital vlsi design. The verification task has been mostly done using functional directed tests and using randomized test. There are tools that can formally prove that the design has no bugs, but it still depends on engineers to define the behavior and it becomes computationally unfeasible very quickly. Every stage of verification flow has been dependent on manual labor and has not been automated. In this work, I present a machine learning based approach to address this problem.

I. INTRODUCTION

After the late 80s, digital designs have become so complicated that gradually, number of verification engineers per design engineer increased to 2. Most of the verification process consists of creating directed tests for specific functionality and creating random tests to test interaction. Most of the bugs found are related to misconnected/unconnected wires and basic functional problems that require very little change in the design. For this reason, I wanted to evaluate if there is any pattern that I can relate to these kinds of bugs, and if we could automatically find them. Outline of the method tried is given below:

Input: Verilog source code before and after bug fixes and final Verilog source code after verification process

Parameters: Number of features to be extracted and length of n-grams

1) For each bug fix:
    a. Parse the Verilog source code and get parse trees before and after the fix
    b. Traverse 2 trees simultaneously (depth first traversal) and extract feature for each node
    c. If features extracted are different place them into sample set
    d. If features extracted are same continue traversal with the children of current node, according to depth first traversal algorithm.
2) Traverse the parse tree of final Verilog source code and extract features in random places
3) Train the classifier using subset of features
    a. Use naïve Bayesian
    b. Use SVM

Output: Prediction results on test set

II. FEATURE EXTRACTION

When traversing a node, I generate the feature vector using

1) Breath-first search order (used in Naïve Bayesian)
   After vectorizing the node types in Breath-first search order, collect number of each n-gram and vectorize the result. Breath-first search order is used for NB

since the n-grams need to reflect dependency of logic that interact

2) Depth-first search order (used in SVM)
   Vectorize the node types with Depth-first search order

module aa(input wire a, input wire b, output reg c); assign c= a&b; endmodule	Module |--ModuleIdentifier | |--IdentifierDefinition |--PortDeclaration | |--DataType | |--Variable | |--IdentifierDefinition | |--IdentifierDefinition |--PortDeclaration | |--DataType | |--Variable | |--IdentifierDefinition | |--IdentifierDefinition |--PortDeclaration | |--DataType | |--Variable | |--IdentifierDefinition | |--IdentifierDefinition |--ContinuousAssign |--NetRegisterAssignment |--IdentifierReference |--BinaryOperator |--IdentifierReference |--IdentifierReference

Table 1-Sample Verilog source code and corresponding parse tree(parse tree node types omitted)

III. DATA ACQUSITION

I used a commercial System-Verilog parser to parse the source code. The parse tree structure is written to a file a custom python script extracts the features from the parse tree. I used scikit-learn[1] python package for training and testing Naïve Bayesian and SVM classifiers.

The Verilog source code is selected carefully to only include simple bugs. (Bugs that do not require more than 2 lines change). I used 80 bugs (160 sample vectors, half with bug, half without bugs) and added 40 vectors from final source code. I did only include single module(Verilog module) parse trees since inter-module interactions complicate the parse tree structure.

Parameters are selected as follows: Number of features = 200, Length of n-grams =4

IV. RESULTS

I used 80% hold out cross validation to measure the performance of the algorithms.

	Real Bug	No-Bug
Predicted Bug	45	11
Predicted No-Bug	19	85

Table 2- Confusion matrix for training set (Naïve Bayesian)

	Real Bug	No-Bug
Predicted Bug	9	6
Predicted No-Bug	7	18

Table 3- Confusion matrix for test set(Naïve Bayesian)

	Real Bug	No-Bug
Predicted Bug	50	9
Predicted No-Bug	14	87

Table 4- Confusion matrix for training set (SVM, Gaussian with gamma=10^{-4} and C=10)

	Real Bug	No-Bug
Predicted Bug	9	5
Predicted No-Bug	7	19

Table 5- Confusion matrix for test set (SVM, Gaussian with gamma=10^{-4} and C=10)

	Percentage (%)-NB	Percentage (%)-SVM
Training Accuracy	81.25	85.63
Test Accuracy	67.50	70.00
Test Recall	56.25	56.25
Test Precision	60.00	64.29
Test Specificity	75.00	79.17

Table 6 – Performance Metric For the Test Set

Since the number of samples is too low, the performance metrics had close to 17% difference between the best and worst test accuracy, depending on random test set selection. For this reason, an average case is chosen.

V. DISCUSSION AND FUTURE WORK

Since the number of samples is too low, testing accuracy varies a lot depending on random test set selection. But, even in the worst case, training accuracy was better than 60% and in the best case, it was close to 80% in both frameworks.

Since the sample set is chosen carefully (restricted to state machine and connection bugs), the algorithms could learn from data. Once the restrictions on how the samples are collected are removed, the training and test accuracy can reduce down to just above 50%. (I tried it with random 100 bugs, which 200 test vectors, with 80% hold out cross validation.).

Apart from these restrictions, the results seem promising and it looks like more data may help generalize the model to a very general bug finder tool.

In this project I restricted the values to parse tree node types and did not use what it actually represents. (For example, +,-,* ,…etc, are all taken to be binary operator, so corresponding values in future vectors are the same). So, taking these into account should be tested.

VI. CONCLUSION

In this work, I presented a new approach to digital logic verification, specifically bug finding. The main stages of the algorithm presented are, parsing the source code, feature extraction and classification using Naïve Bayesian or SVM. As in any learning problem, feature extraction is the most critical step. Results show that it is promising but, may need a larger scale testing with far more data and more complicated features.

REFERENCES

[1] http://scikit-learn.org