Programming Hints Generation based on Abstract Syntax Tree Retrieval

Han Wan, Hongzhen Luo, Zihao Zhong, Xiaopeng Gao
School of Computer Science and Engineering
Beihang University
Beijing, China
{wanhan, luohz, zhongzihao, gxp}@buaa.edu.cn

***Abstract*—This paper presents research that Works in Progress (WIP). Small private online courses (SPOCs) have recently received extensive attention in computing education. In SPOCs, programming exercises are frequently included to train students' programming skills. Abstract Syntax Tree Retrieval (ASTR) is a system that can help students solve Python problems by inferring the coding goals. However, the coding goal retrieved by ASTR gives students little information about what to do next. In response to this limitation, this work focuses on generating modification hints for students based on the coding goal. In addition, this paper reports on an effort to translate this idea over to Verilog-HDL programming problems. Without any programmed expert knowledge, the final results demonstrate that our system is generally accurate for 1 out of 2 submissions to give hints at a minimum. And for some favorable problems, it potentially performs much better. Furthermore, the results indicate that in the process of retrieval, weighted tree edit distance calculations resulted in improved accuracy over metric tree edit distance calculations.**

Keywords—Program evaluation, Online, Experiential learning, Computer engineering

I. INTRODUCTION

Small private online course (SPOC) is characterized by improving teaching effectiveness, and gradually becoming the mainstream online education model of higher education. In SPOCs, laboratory programming exercises are frequently included to improve students' programming skills. Simultaneously, the online judge is used to verify the correctness of students' programs.

For a long time, in the field of computer education, we can provide students with feedback to a certain extent. The compiler can point out students' syntax errors. However, these feedbacks have great limitations. Compiler messages are notoriously unhelpful and the online judge is based on test cases, which only provide information on whether to pass or not. The burden falls on the teaching assistants to explain to the students where they made mistakes. However, with the increase in the number of students, the work pressure on teaching assistants will be too great to bear [1].

Intelligent Tutoring System (ITS) can reduce the burden of teaching assistants, and thus has a favorable application prospect in computing education. Abstract Syntax Tree Retrieval (ASTR) [2] is an ITS that uses case-based reasoning to infer students coding goals from previous solutions to Python coding problems. In the field of computer science, edit distance is a well-known method for determining the similarity of hierarchical structure [3][4][5]. And the technique in ASTR is based on a tree edit distance algorithm working with abstract syntax trees (ASTs). The retrieved result of ASTR has the smallest tree edit distance and the highest similarity. When a submission has not solved the problem, ASTR is used to calculate and return the most similar existing solution, i.e., the coding goal.

However, the coding goal retrieved by ASTR gives students little information about program modifications. Driven by this limitation, based on the coding goal retrieved by ASTR, this work focuses on generating modification hints for students. And implementing this idea to Verilog-HDL programming problems. More specifically, modifications made by the coding goal may contain hints and our focus is on how to extract these hints.

To achieve this goal, it is essential to analyze the fine-grained sequence of code changes between the coding goal and its latest wrong version. Pyverilog [6] is an open-source toolkit that can parse Verilog-HDL code strings into ASTs. In this paper, Pyverilog is used to build the ASTs. Then Gumtree [7], another open-source tool, is applied to these ASTs to calculate fine-grained and accurate edit scripts. Nevertheless, the edit scripts generated by Gumtree are low-level granularity, and the edit actions are at the level of AST nodes instead of expressions. The low-level edit scripts make the follow-up modifications analysis difficult. Therefore, a set of processing steps include clustering the scattered low-level AST nodes are applied to extract more understandable information from the edit scripts produced by Gumtree, and finally generate hints.

II. RELATED WORK

Our research builds on previous work in many fields. The most important of these include: AST, fine-grained code differencing, ASTR, and bug fix pattern.

A. Abstract Syntax Tree

An AST is a tree representation of the abstract syntax structure of a program, where each AST node represents a grammar element. Pyverilog is an open-source tool that parses Verilog-HDL code strings into ASTs. There are four key libraries in Pyverilog: (1) parser, (2) dataflow analyzer, (3) control-flow analyzer, and (4) Verilog-HDL code generator. In this paper, Pyverilog is used as a parser for syntax parsing.

B. Fine-grained Code Differencing

Fine-grained code differencing is a hot research topic. Reference [8] proposes a method that can be used for generating accurate and compact edit scripts. A novel approach was presented for hyperparameter optimization of AST differencing in [9]. In order to reduce the time cost of calculating the difference between the ASTs, a feasible way is to reduce the amount of calculation. Reference [10] proposed a novel approach to prune the AST with hunks and this approach can speed up tree differencing.

Gumtree is a fine-grained code differencing tool based on AST. The edit scripts generated by Gumtree contain four types of edit actions: *update*, *add*, *move* and *delete*.

The edit actions in edit scripts are at the level of AST nodes, which is low-level granularity. To address this limitation, a set of preprocessing steps in [11] was used to generate more understandable code differences from the edit scripts generated by Gumtree.

C. Abstract Syntax Tree Retrieval

ASTR is an ITS, which uses case-based reasoning to suggest the coding goals. In ASTR, the similarity between two ASTs is calculated with the Zhang Shasha (ZSS) tree edit distance algorithm [12]. Similar to the idea of ASTR, this paper uses the edit scripts generated by Gumtree to obtain similarity.

D. Bug Fix Patterns

In order to reduce bugs in software projects, a key step is to identify bug fix patterns. Reference [13] identifies 38 bug fix patterns and exposes 37 new patterns of nested code structures, which frequently host the bug fix edits. Reference [14] found 27 common bug fix patterns in Java and these bug fix patterns are automatically extracted from seven Java software projects.

Reference [15] explored Verilog-HDL programming language, and 25 bug fix patterns are observed in four open-source Verilog-HDL projects. Those 25 bug fix patterns are divided into 6 categories: *IF-RELATED*, *MODULE-RELATED*, *CASE-RELATED*, *ALWAYS-RELATED*, *ASSIGNMENT-RELATED*, and *CLASSFIELD-RELATED*. To better satisfy our research, according to the teaching experience, we extend these 6 categories by adding *LOOP-RELATED*. And these 7 categories are used as hints for students.

III. Method

The goal of our system is to give students hints when they get stuck. Fig. 1 shows this situation. The left-hand side code fragment contains a bug that causes the program to fail the tests. After a long period of debugging, the student modified the *if statement* and added an *assignment statement*, and finally passed the tests. Ideally, our system should give the student modification hints related to *if statement* and *assignment statement*. To achieve this, we need to first find the coding goal that is most similar to the current student's code, and then extract hints from the modifications of the coding goal.

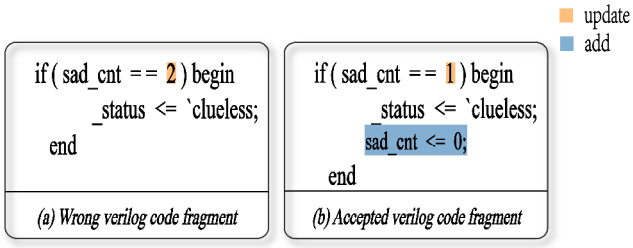

Fig. 1. An example of code modifications.

A. Retrieve The Coding Goal

To generate modification hints, the coding goal should be obtained first. This paper migrates the idea of ASTR to Verilog-HDL programming language. The similarity is calculated by the edit distance between ASTs and the AST with the highest similarity is the retrieved result of ASTR.

Gumtree has a built-in Verilog-HDL parser to build ASTs. However, the ASTs generated by this built-in parser are complex. In comparison, Pyverilog can generate more concise ASTs. Therefore, this work uses Pyverilog as the parser to parse Verilog-HDL code strings into ASTs. For the wrong code fragment in Fig. 1, the AST generated by Pyverilog is shown in Fig. 2. And the accepted code fragment in Fig. 1 is parsed into the AST in Fig. 3.

After the ASTs are built, Gumtree is applied to generate the edit scripts between these syntax trees. The edit scripts generated by Gumtree contain four types of edit actions: *update*, *add*, *delete* and *move*. For example, by using Gumtree to calculate the difference between the two ASTs in Fig. 2 and Fig. 3, the edit script in TABLE I can be obtained. Every node in the AST is identified by a unique number. In the edit script in TABLE I, the value of node 185 in Fig. 2 is updated to 1. Four nodes (192 to 195) are added to the AST in Fig. 2, forming the AST in Fig. 3.

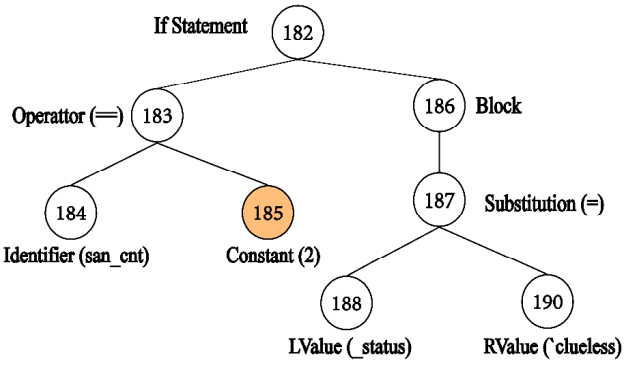

Fig. 2. The AST of the wrong code fragment.

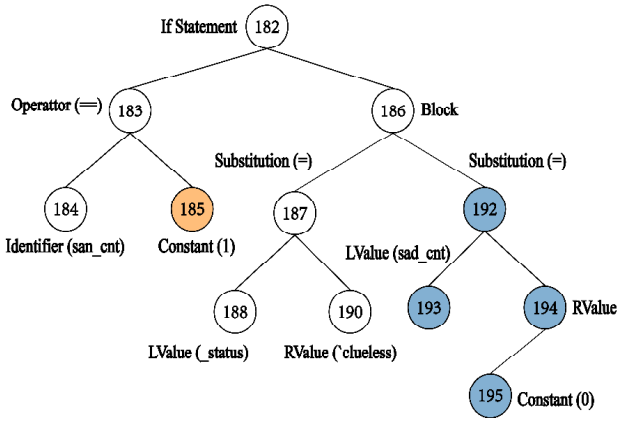

Fig. 3. The AST of the accepted code fragment.

TABLE I. The Edit Script Generated by Gumtree

Index	Edit Action
1	update(185,1)
2	add(192,186,1,Substitution,=)
3	add(193,192,1,LValue,sad_cnt)
4	add(194,192,2,RValue,null)
5	add(195,194,1,Constant,0)

This research uses the edit scripts to calculate similarity. For our experiments, we use three different weightings for the costs of edit actions. The first weighting assigns the cost to each edit action as 1, which is referred to as the metric tree

edit distance calculation. The second weighting adopts the weight value recommended by [2]. The design of the third weighting is determined according to the proportion of edit actions. Eventually, the code with the highest similarity is chosen from all the candidate solutions as the coding goal.

B. Extract Hints

In this part, modifications are extracted from the difference between the coding goal and its latest wrong version. Similar to the previous process, we first use Pyverilog to build the ASTs. Then Gumtree is applied to generate the edit script of the ASTs.

However, the edit scripts generated by Gumtree are low-level granularity, and the edit actions are at the level of AST nodes instead of expressions. To generate modification hints, changes in the expression level need to be observed. For example, desired hints in Fig. 1 are related to *if statement* and *assignment statement*, while the actions in the edit script focus on the nodes in the AST. Therefore, it is necessary to extract expression-level information from the edit scripts to generate hints.

1) Define Node Types

Adopting the terminology used by [11], we define these expression-level nodes as *base* nodes and other nodes as *composite* nodes. A *base* node is recognized as a node that involves a high-level AST element (declaration or statement). And a *composite* node does not hold statements or declarations.

Reference [15] found 25 common bug fix patterns in four open-source Verilog-HDL projects, and these bug fix patterns are divided into 6 categories. Loop-related bug fix pattern does not appear in any of these 6 categories. However, according to our teaching experience, for basic problems, mistakes related to loops sometimes occur. To better satisfy our research, we extend the 6 categories in [15] by adding *LOOP-RELATED*. And these 7 categories are used as hints for students. In this paper, the *base* nodes are combined with the corresponding hints in TABLE II. Once the edit action about the *base* node is observed, the corresponding hint can be given automatically.

TABLE II. THE COMBINATION OF NODE TYPES AND HINTS

Base Node In Pyverilog	Related Hint
If Statement	IF-RELATED
Module Definition	MODULE-RELATED
Assign	ASSIGNMENT-RELATED
Block	
Substitution	
Case Statement	CASE-RELATED
Case	
Always	ALWAYS-RELATED
Decl	CLASSFIELD-RELATED
For Statement	LOOP-RELATED
While Statement	

2) Establish Mapping

The purpose of establishing mapping is to extract more understandable information through clustering the scattered low-level AST nodes. For low-level nodes in edit scripts, the high-level information corresponding to them needs to be mapped. We travel the root node in a depth-first way to establish the mapping between the child node and its parent node. The generated mapping will be used later.

3) Process the Edit Script

In this part, the mapping between the child node and its parent node is used to generate hints. Unlike [10], which only focuses on *add* and *delete* operations, this work also focuses on *update* and *move* operations. For every edit action in the edit scripts, if the target node is a *base* node and has not been visited, the corresponding hint in TABLE II is added to the answer set. Otherwise, the system recursively lookup the parent node until it is a *base* node. If the ancestor node has not been visited, the corresponding hint is added to the answer set.

Take the edit script in TABLE I as an example. The first edit action *update* (185, 1), acts on node 185 in Fig. 2. Node 185 is not a *base* node. Then our system recursively lookup the parent node until it is a *base* node. Node 182 is a *base* node and is the nearest ancestor of node 185. Because the node type of node 182 is *If Statement*, *IF-RELATED* in TABLE II is added to the answer set and node 182 is marked as visited. The second edit action *add* (192, 186, 1, Substitution, =), acts on node 192 in the AST in Fig. 3. The node type of node 192 is *Substitution*, which is a *base* node. So *ASSIGNMENT-RELATED* is added to the answer set and node 192 is marked as visited. The third edit action *add* (192, 192, 1, LValue, sad_cnt) acts on node 193, and node 193 is not a *base* node in the AST in Fig. 3. Node 192 is found recursively. But node 192 has been visited, the third edit action in the edit script will be ignored. The fourth and fifth edit actions are processed similarly to the third edit action. Finally, the edit script in TABLE I produces two hints: *IF-RELATED* and *ASSIGNMENT-RELATED*.

IV. DATASET

Our research uses a dataset to test the effectiveness of our system. The dataset consists of submissions created by students to solve basic problems in the computer structure course. The submissions are written in Verilog-HDL programming language.

There are a total of 475 successful submissions, in 15 subproblems of P_1 (P_1 is the warm-up programming exercise in this course), as illustrated in TABLE III. The level of these programming questions is targeted to the beginning programmers.

TABLE III. OVERVIEW OF THE DATASET

Problem ID	# of Codes	Problem ID	# of Codes
270	30	426	1
32	30	504	38
326	4	506	120
328	27	508	50
394	5	509	139
403	3	510	18
423	3	520	6
425	1		
Total # of codes			475

V. RESULTS AND DISCUSSION

Bug fix patterns are important to generate hints. In our experiment, the bug fix patterns of the 475 codes are explored first. The bug fix patterns and their repetitiveness are shown in TABLE IV. And the following conclusions can be drawn from the table:

- For basic Verilog-HDL programming exercises in this course, *ASSIGNMENT-RELATED* modifications account for the largest proportion, reaching about 68%.

- The modifications related to *assignment* and *if* statements account for the vast majority, while the other five bug fix patterns account for a small proportion.

The intention of our experiments is to test the accuracy of the hints generated by our system. To simulate ideal conditions, the problems that contain too few codes were eliminated. For each problem, we randomly use 80% of the codes as the candidate codes and the remaining 20% of the codes are used to test the accuracy. For each wrong code, the system will give modification hints and these hints will be reviewed by a human expert. The expert gives a score, which is in the range of 0 to 1, to the generated hints. 1 means that these hints can exactly help students modify their programs. On the contrary, 0 means that these hints can hardly provide help. The accuracy of a problem is the overall score of the test codes compared to the total number of the test codes, in the form of a percentage.

TABLE IV. CONCISE BUG FIX PATTERN

Bug Fix Pattern	Repetitiveness
ASSIGNMENT-RELATED	7275(68.62%)
IF-RELATED	1892(17.85%)
CASE-RELATED	809(7.63%)
CLASSFIELD-RELATED	264(2.49%)
ALWAYS-RELATED	166(1.57%)
MODULE-RELATED	106(1.00%)
LOOP-RELATED	90(0.85%)
Total	10602(100.00%)

For the similarity calculation in our experiment, three different weightings for the costs of edit actions are used. The first weighting assigns the cost to each edit action as 1. We refer to this as metric weighting. The result shows that the accuracy of hints is affected by different problems, from 50% to 83%. Question 509 contains the largest number of test cases but still performed well, as depicted in TABLE V.

TABLE V. PERFORMANCE OF METRIC WEIGHTING

Problem ID	# of Candidate Codes	# of Test Codes	Accuracy
32	24	6	50%
270	24	6	67%
328	21	6	75%
504	30	8	83%
506	96	24	58%
508	40	10	77%
509	111	28	73%
510	14	4	61%

The second weighting adopts the weight value recommended by [2], which places a weight of 3 on the edit action *add*, a weight of 2 on *update* and *move*, and a weight of 1 on *delete*. The result is shown in TABLE VI. It can be concluded that the second weighting improves the accuracy of our system. Especially for question 510, the accuracy increased by 34% and reached 95%.

However, the second weighting does not consider the proportion of edit actions. The design of the third weighting is determined according to the proportion of edit actions. According to the proportion of each action, we set the weight value of *add* to 8, the weight value of *update* and *move* to 3, and the weight value of *delete* to 12. Compared to TABLE VI, performance using the third weighting improved the accuracy on almost all the questions (except for question 510), as shown in TABLE VII.

TABLE VI. PERFORMANCE OF WEIGHTING RECOMMENDED BY ASTR

Problem ID	# of Candidate Codes	# of Test Codes	Accuracy
32	24	6	50%
270	24	6	67%
328	21	6	85%
504	30	8	84%
506	96	24	68%
508	40	10	79%
509	111	28	80%
510	14	4	95%

TABLE VII. PERFORMANCE OF WEIGHTING SET ACCORDING TO THE PROPORTION OF EDIT ACTIONS

Problem ID	# of Candidate Codes	# of Test Codes	Accuracy
32	24	6	57%
270	24	6	68%
328	21	6	88%
504	30	8	85%
506	96	24	79%
508	40	10	86%
509	111	28	84%
510	14	4	78%

VI. CONCLUSION AND FUTURE WORK

ASTR is interested in retrieving the coding goal, while this work focuses on generating hints for students based on the coding goal. In this work, the fine-grained sequence of code changes between the coding goal and its latest wrong version is generated. Then the code differencing is used to produce modification hints for students. Same as ASTR, our system requires no information about the programming problems.

The result achieved by our system is encouraging. For basic Verilog-HDL programming problems, the modifications related to *assignment* and *if* statements account for the vast majority. Our system is generally accurate for 1 out of 2 submissions to give hints at a minimum, with favorable problems potentially performing much better. For calculating similarity, weighted tree edit distance result in improved accuracy over metric tree edit distance. In addition, the accuracy of the system can be further improved by setting appropriate weight values.

However, programming hints generated by this research are only for basic problems in the computer structure course. For complex problems in this course, further exploration is needed. In addition, in order to have better universality, the hints generated in this paper are included in seven categories. For a specific program, the generated hints are relatively vague. How to make a trade-off between accuracy and universality is what we need to do next. In this paper, the proportion of edit actions is used to set the weight value to improve the accuracy of the system. In future work, more strategies for setting weight values will be explored.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (61907002).

REFERENCES

[1] K. Rivers and K. R. Koedinger, “Automatic generation of programming feedback: A data-driven approach,” in The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013), vol. 50, 2013, pp. 50–59.

[2] P. Freeman, I. Watson, and P. Denny, “Inferring student coding goals using abstract syntax trees,” in Case-Based Reasoning Research and Development, 2016, pp. 139–153.

[3] P. Bille, “A survey on tree edit distance and related problems,” Theoretical computer science, vol. 337, no. 1-3, pp. 217–239, 2005.

[4] R. Yang, P. Kalnis, and A. K. Tung, “Similarity evaluation on tree-structured data,” in Proceedings of the 2005 ACM SIGMOD international conference on Management of data, 2005, pp. 754–765.

[5] M. Pawlik and N. Augsten, “Tree edit distance: Robust and memoryefficient,” Information Systems, vol. 56, pp. 157–173, 2016.

[6] S. Takamaeda-Yamazaki, “Pyverilog: A python-based hardware design processing toolkit for verilog hdl,” in Applied Reconfigurable Computing, 2015, pp. 451–460.

[7] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Finegrained and accurate source code differencing,” in Proceedings of the International Conference on Automated Software Engineering, 2014, pp. 313–324.

[8] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, “Generating accurate and compact edit scripts using tree differencing,” in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 264–274.

[9] M. Martinez, J.-R. Falleri, and M. Monperrus, “Hyperparameter optimization for ast differencing,” 2020. [Online]. Available: https://arxiv.org/abs/2011.10268

[10] C. Yang and E. J. Whitehead, “Pruning the ast with hunks to speed up tree differencing,” in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 15–25.

[11] O. Meqdadi and S. Aljawarneh, “A study of code change patterns for adaptive maintenance with ast analysis,” International Journal of Electrical and Computer Engineering (IJECE), vol. 10, pp. 2719–2733, 2020.

[12] K. Zhang and D. Shasha, “Simple fast algorithms for the editing distance between trees and related problems,” SIAM Journal on Computing, vol. 18, no. 6, pp. 1245–1262, 1989.

[13] M. R. Islam and M. F. Zibran, “How bugs are fixed: Exposing bug-fix patterns with edits and nesting levels,” in Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 1523–1531.

[14] K. Pan, S. Kim, and E. J. Whitehead, “Toward an understanding of bug fix patterns,” Empirical Software Engineering, vol. 14, no. 3, pp. 286–315, 2009.

[15] S. Sudakrishnan, J. Madhavan, E. J. Whitehead Jr, and J. Renau, “Understanding bug fix patterns in verilog,” in Proceedings of the 2008 international working conference on Mining software repositories, 2008, pp. 39–42.