Analysis of Logic Errors Utilizing a Large Amount of File History During Programming Learning

Katsuyuki Umezawa
Depertment of Information Science
Shonan Institute of Technology
Kanagawa, Japan
umezawa@info.shonan-it.ac.jp

Makoto Nakazawa
Department of Industrial Information Science
Junior College of Aizu
Fukushima, Japan
nakazawa@jc.u-aizu.ac.jp

Manabu Kobayashi
Center for Data Science
Waseda University
Tokyo, Japan
mkoba@waseda.jp

Yutaka Ishii
Faculty of Education
Chiba University
Chiba, Japan
yishii@chiba-u.jp

Michiko Nakano
Faculty of Education and Integrated Arts and Sciences
Waseda University
Tokyo, Japan
nakanom@waseda.jp

Shigeichi Hirasawa
Research Institute for Science and Engineering
Waseda University
Tokyo, Japan
hira@waseda.jp

***Abstract*—We proposed an editing record visualization system that can confirm learner modification of programs by storing a learning log. This system was utilized for an actual flipped classroom and stored an enormous volume of learning logs. Each learning log contained all the source code being modified until the program was completed. We also developed a debugging exercise extraction system to automatically generate problems with syntax errors for debugging practice using these learning logs. In this paper, we propose a method of identifying logic errors in cases where no error information is obtained by analyzing the learning log.**

Keywords—programming, logic error, editing record, learning log, debugging practice

I. INTRODUCTION

Understanding the programming learner source-code writing process, anticipating the kinds of challenges encountered, and the method of overcoming these challenges are quite difficult. Source code file version history is insufficient in achieving this understanding, but taking a detailed record of a user's operation can be effective [1][2][3]. Previous research proposed a system for visualizing editing history that can serve as an environment for learning about programming language learning [4]. This system both prepares the learning environment and confirms the learning situation. Additionally, this system accumulates learning logs, and hence program modifications by the learners can be visualized. Programming languages that can be used with this editing history visualization system include (for example) C/C++, Java, JavaScript, and Scratch, and this system can also be applied to writing in English. In previous works, we proposed an effective flipped classroom, referred to as a "grouped flipped classroom," which is based on self-study log information [5][6]. We utilized the editing history visualization system when this classroom was applied to an actual lesson [7][8]. As a result, numerous learning logs were accumulated when a total of approximately 600 students took classes for eight weeks in three years.

We conducted a programming lesson and found that several students asked faculty members for help without reading the error messages that were displayed. The editing history visualization system accumulates all the source code of the process being modified until the program is completed. We thought that by automatically extracting the source code containing errors, we could help learners to practice mistake correction. Similarly, we developed a tool for realizing this extraction [9][10].

In contrast, in this work, we use the same learning history to analyze logic errors. Logic errors are non-syntactic errors, and hence the compiler outputs no error information, leading to difficulty in automatic detection of these errors. A previous study [11] reported that some types of logic errors were counted by observing the learning log. However, when many learning logs are targeted, human observation of the process is difficult. In this work, we propose a method for automatically detecting logic errors by accumulating and analyzing a large amount of programming learning history.

In Section 2, we describe related work and provide an outline of the editing history visualization system. In Section 3, we explain the operation of a previous algorithm that we developed to identify syntax errors. In Section 4, we modify the algorithm to identify logic errors. The experimental method and analysis of the results are presented in Section 5. In addition, the conclusion is presented in Section 6.

II. RELATED WORK

A. Learning History Accumulation System

Understanding how programming learners write source code, the problems encountered during this writing, and the methods of solving these problems is difficult. Previous studies [1][2] used accumulated learning history to propose functions for displaying the operation history, setting and filtering display conditions, and displaying or restoring source code from any past state. A previous study [3] reported that feeding the good and bad points back to the learner by accumulating and analyzing the detailed learning history in real time is possible. Additionally, the system in that study contained a function that allows teachers to identify areas of difficulty for individual learners. However, the history of all accumulated learners has only been analyzed manually, and improvement in quality, based on the analysis results, has yet to be achieved.

B. *Learning History Visualization System*

In a previous work, we proposed an editing history visualization system that serves as a learning environment for programming language learning that is based on the same concept as the system shown in section II.A [4].

The editing history visualization system is a learning environment specialized for programming language learning beginners and is also applicable to English language learning. This system facilitates preparation of the learning environment and understanding of the learner situation. Regarding preparation of the learning environment, installation of a development environment is unnecessary, and this system can be used with only a browser. Learners will therefore be able to use the system from a PC, smartphone, or tablet. In addition, with regard to understanding the situation of the learner, the difference between the current source code and the previous state of the execution result can be displayed on the teacher's screen. Looking at this difference allows confirmation that the learner modified their code. Additionally, the questions each learner is solving can be visualized. Programming languages that can be used with this editing history visualization system include (for example) C/C++, Java, JavaScript, and Scratch, and the system also considers writing problems in English.

Figure 1 shows the editing history visualization system for the Java programming language. Figure 2 shows the teacher's confirmation screen.

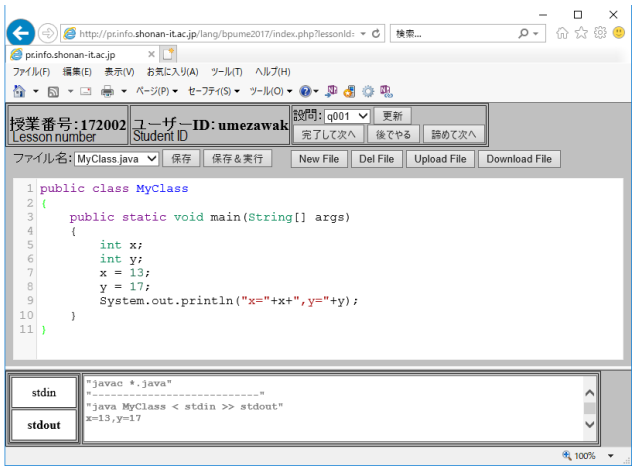

Fig. 1. Editing history visualization system (screen for learner)

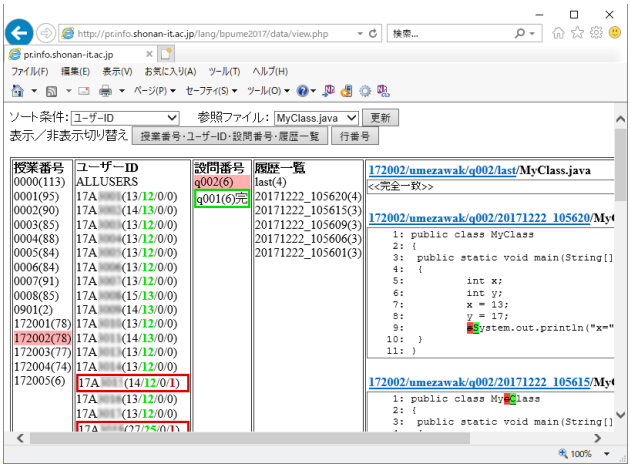

Fig. 2. Editing history visualization system (screen for teacher)

III. Debugging Exercise Extraction Sysytem

We developed a debugging exercise extraction system that automatically extracts syntax errors [9][10]. In this section, we briefly describe a debugging exercise extraction system for syntax errors.

A. *Overview of Our Proposed System*

Figure 3 shows the overall configuration of the debugging exercise problem extraction system for syntax errors. As shown in the figure, the existing editing history visualization system is used until the learning history is accumulated. The debugging exercise extraction tool refers to the learning history accumulated by the visualization system, compares the source code containing errors with the correct source code, and extracts the source code containing errors. The extracted code is then distributed in a modified version of the visualization system.

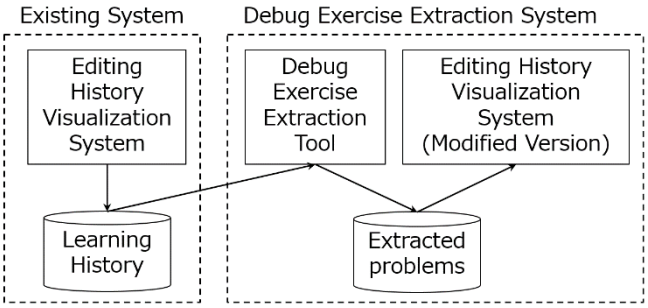

Fig. 3. Overall configuration of the debugging practice problem extraction system

B. *Debugging Exercise Extraction Tool*

The debugging exercise extraction tool compares the learning history accumulated by the editing history visualization system, specifically, MyClass.java existing in the complete folder (last) for each problem, and MyClass.java in the process of coding. The tool extracts the number of mistakes and the number of misspelled characters in each mistake, and then reconstructs the folder based on these extractions. Two types of source code are considered: one containing an error to be corrected and another without errors. Only the source code containing the error is extracted.

C. *Extraction Algorithm*

The extraction algorithm for this tool consists of the following steps.

- Repeat the following for the entire history of the folder configuration in Figure 4.
- Check if last.info contains "end."
- For folders other than the last folder, check whether "errors" are described in the stdout file.
- Find differences between MyClass.java in the folder (other than the last folder that describes the error) and MyClass.java in the last folder (In the Java version of the editing history visualization system, the class name containing the main function is MyClass, and its file name is specified as MyClass.java).
- At that time, the system counts the number of mistakes in a source code and the number of characters included in each mistake.

- The MyClass.java file is copied following the folder structure shown in Figure 5, depending on the number of mistakes and the number of misspelled characters contained in each mistake.

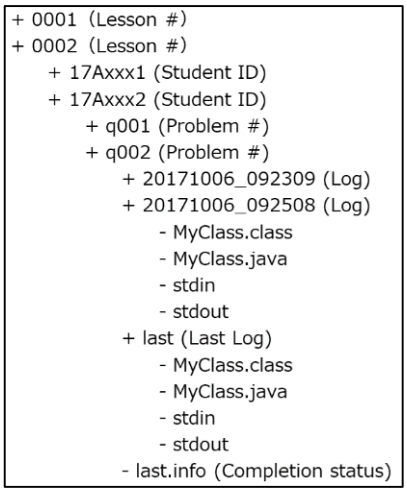

Fig. 4. Folder structure of editing history visualization system

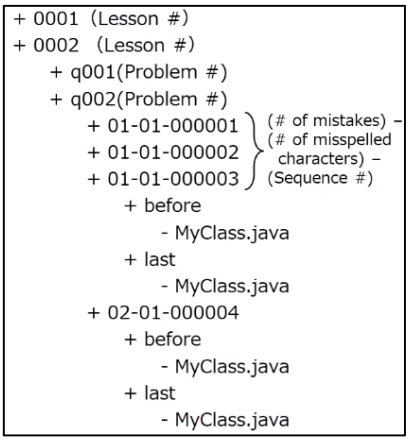

Fig. 5. Folder structure of debugging exercise extraction tool

IV. Extend to Handle Logic Errors

A. Overview

The algorithm shown in section III.C contains a step to confirm whether the "error" is explicitly described in the program learning history. This checking allows identification of the program source file causing the syntax error. However, in this case we want to analyze logic errors.

The above algorithm targets syntax errors, and hence handles the set of program source files in which "errors" are explicitly described in the learning history. In contrast, the set of program source files for which no "error" description is provided in the learning history includes program source files, some of which include logical errors. Of course, the set also includes source files that are just edited, without logical errors. Consider the case that the correct answer must be provided in the limited time of a university class. In this case, the editing work is considered the task of removing some logical error (future analysis is needed to determine the number of non-logical errors included in the set). Identifying logic errors by analyzing these source files may be possible. Syntax errors are excluded from the set of program source files that provide no "error" description in the learning history.

B. Consideration of Output Results

We changed the algorithm of the existing debugging exercise extraction tool to output the difference between a source file where no "error" description is provided in the learning history and the final source file. Examples of the output results are shown in Figures 6–8. Figure 6 shows the modification of the character string enclosed in double quotes, which may be treated as a logical error depending on the task (rather than a syntax error). Figure 7 shows the modification performed on the conditional part of the if statement, and Figure 8 shows the modification of the substitution statement. These are also classified as logic errors rather than syntax errors.

```
 1: public class MyClass
 2: {
 3:     public static void main(String[] args)
 4:     {
 5:         System.out.println("Hello World");
 6:     }
 7: }
```

Fig. 6. Modification performed on string

```
 1: import java.io.*;
 2:
 3: public class MyClass
 4: {
 5:     public static void main( String[] args ) throws IOException
 6:     {
 7:         int x,y;
 8:         x = 5;
 9:         y = 3;
10:         if(x>=y){
11:             System.out.println( + x +"は" + y + "より大きい");
12:         }
13:     }
14: }
```

Fig. 7. Modification performed on conditional part of if statement

```
 1: import java.io.*;
 2:
 3: public class MyClass{
 4:     public static void main(String[] args) throws IOException{
 5:         int x, y ,z;
 6:
 7:         x=3;
 8:         y=7;
 9:
10:         z=x+y;
11:         x=y=z=x;
12:         y=x=z;
13:         System.out.println("x="+ x +", y="+ y);
14:     }
15: }
```

Fig. 8. Modification performed on substitute statement

C. Type of Logic Error

As shown in the previous section, we found that the modifications contained some types of logic errors. In addition to these, we found that the modifications contained the logic errors in the learning history shown in Table I.

TABLE I. Type of Logic Error

Type	Description
Spaces	Add or delete spaces or tabs.
Comments	Add or delete comments by //.
Strings	Change the character string enclosed by " and "
Brackets	Add or delete (,),{, and }.
For statements	Modification of for statement itself and conditional part.
While statements	Modification of while statement itself and conditional part.
If statements	Modification of if statement itself and conditional part.
Else statements	Modification of else statement itself.
Println	Rewrite println, printf, print, and modification contents of ().
Semicolons	Add or delete;.
Arrays	Modification of array size and index.

Type	Description
Spaces	Add or delete spaces or tabs.
Variables	Modification of variable.
Numerics	Modification of numeric number.
Substitution statements	Modification of substitution statement.
Expressions	Modification of expression.
Other	Modifications that cannot be classified as above.

D. Debugging Exercise Extraction Tool for Logic Errors

We visually confirmed the modifications of the source files in our analysis. However, manually processing large amounts of data is impossible, and hence we added a function to the existing debugging exercise extraction tool that automatically classifies logic error types (see Figure 9).

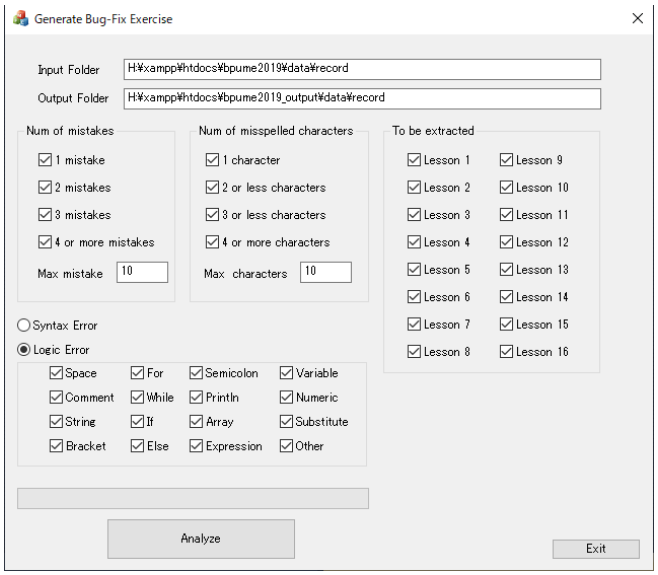

Fig. 9. Debugging exercise extraction tool for logic errors

E. Identification of the Logic Errors

Basically, our tools are based on Diff, i.e., an algorithm that finds the difference between two documents (see Fig. 7 for example). In the first step of the process, the modified part is identified with Diff. The left side of the modified part was searched for a character string "if (." Next, search the right side of the modified part and find that there is a ")" before the line feed code appears. Therefore, the modified part corresponds to the conditional part of the if statement. For other types of logic errors, basically, the logic error is found by searching the left and right sections of the modified part.

V. Experiment and Analysis

A. Class that Collected Data

We proposed a new flipped classroom method [5][6] that divides students into three groups before each class based on their e-learning self-study logs and level of understanding. The three groups consist of students who studied the lesson and fully understand its contents, students who studied the lesson but only partly understand its contents, and students who did not study the lesson and thus do not understand its contents. We applied this grouped flipped classroom method to 16 weeks of actual lessons in the autumn semesters of 2017, 2018, and 2019 at our university. Furthermore, we showed the effectiveness of this method in terms of understanding after self-study and understanding after face-to-face instruction [7][8]. The application of this flipped classroom method utilized the editing history visualization system for the Java programming language shown in section II.B. In this application, we accumulated many learning logs, with logs from 183 students, 189 students, and 224 students in 2017, 2018, and 2019, respectively. The contents of the Java language learning class are shown in Table II.

TABLE II. Description of Class

Week	Content	Num. of questions
1st week	Java language (Input/Output)	7
2nd week	Java language (Variable/Arithmetic)	12
3rd week	Java language (Branch)	13
4th week	Java language (Repetition)	22
5th week	Java language (Array)	8
6th week	Java language (Method)	8
7th week	Java language (Class I)	6
8th week	Java language (Class II)	4

B. Analysis Results

The number of questions shown in Table II was performed each week during the eight-week class described in Section V.A. Each time the learner presses the "Build & Execute" button when creating a program that solves these problems, the source program at that time is accumulated as a learning history. We thus accumulated 203,436 source files (72,193, 61,721, and 69,522 in 2017, 2018, and 2019, respectively) that were then analyzed.

When solving a task, many learners copy and modify the source code of the previous problem. The expectation is that "Build & Execute" will be performed after the copied source code has been significantly modified. In other words, as compared with the complete version of the source code, modifications in many places or modification of many characters would be unsuitable for logic error analysis. Therefore, we exclude source code from being detected as a logic error when the "number of mistakes" and "number of misspelled characters" are greater than 10. Table III shows the number of detections for each logical error type and the percentage (%) of the total number of detections for each year.

TABLE III. Number of Detections and Percentage for each Logic Error Type

Type	Num. of Detections (Percentage %)		
	2017	2018	2019
Spaces	4189 (19.94)	2735 (15.12)	2934 (14.85)
Comments	124 (0.59)	34 (0.19)	289 (1.46)
Strings	2616 (12.45)	2675 (14.78)	2410 (12.20)
Brackets	1140 (5.43)	1164 (6.43)	1228 (6.21)
For statements	2771 (13.19)	2223 (12.29)	2577 (13.04)
While statements	152 (0.72)	108 (0.60)	152 (0.77)
If statements	1453 (6.92)	1122 (6.20)	1384 (7.00)
Else statements	49 (0.23)	31 (0.17)	183 (0.93)
Println	89 (0.42)	101 (0.56)	93 (0.47)
Semicolons	869 (4.14)	649 (3.59)	649 (3.28)
Arrays	371 (1.77)	333 (1.84)	454 (2.30)
Variables	3215 (15.30)	2476 (13.68)	2390 (12.10)
Numerics	1447 (6.889)	2006 (11.09)	2313 (11.71)
Substitutions	279 (1.33)	252 (1.39)	331 (1.68)
Expressions	2220 (10.57)	2155 (11.91)	2333 (11.81)
Other	26 (0.12)	29 (0.16)	40 (0.20)
Total	21010 (100)	18093 (100)	19760 (100)

We performed χ^2-test to find out if there is a difference in the distribution for each year. We got the result of χ_0^2 = 1102.6, the degrees of freedom=(3-1)×(16-1)=30, p-value < $2.2e^{-16}$. From this result, it was concluded that the distribution of each year is not statistically the same. However, from Table III, the tendency of frequency seems to be similar in each year.

Table III shows that "Spaces" have the highest number of detections; however, these, along with changes to "Strings," are less important (than other logic error types) for programming comprehension. Many detections for "For statements" and "If statements" are related to program control structures. The "While statements," which are also control structures, are used infrequently, and therefore the number of detections is small. However, many detections of changes in "Variables" and "Numbers" are registered. This is attributed to the nature of university lessons, as similar problems are solved successively. Although many "Expressions" are detected, the logic errors contained here may require further analysis. Additionally, the compiler output no error for those logic errors classified as "Other," but source code containing a correction by double-byte characters is detected.

C. Evaluation of Execution Time

We executed the debugging exercise extraction tool on the computer described in Table IV using the extraction options listed in Figure 9. Table V shows the number of source files analyzed in each year and the analysis execution time. The analysis of the entire learning history is only necessary for ~200 learners in a year, and hence this execution time can be considered sufficiently practical.

TABLE IV. SPECIFICATIONS OF THE COMPUTER THAT PERFORMED THE MEASUREMENT

Item	Description
CPU	Intel(R) Core(TM) i5-2400 CPU @ 3.10 GHz
Memory	16 GB
OS	Windows 10 Pro (64 bit)

TABLE V. EXECUTION TIME

Year	Num. of Source File	Execution Time (s)
2017	72,193	290
2018	61,721	249
2019	69,522	272

(Note: Execution time is the average of three measurements)

VI. CONCLUSION AND FUTURE WORK

We were able to detect logic errors from a large volume of programming history where the compiler outputs no error information, owing to the lack of syntax errors. Using these results, we can extract source code containing one logical error related to (for example) the "for" statement. This source code can be presented to students, who can practice debugging to correct logic errors.

For example, consider a logic error related to a variable, a variable related to a double loop can be easily mistaken. Variables used for iteration and those used for conditional branching within that iteration are also mistaken. In the future, we would like to be able to automatically perform a systematic analysis, by combining control statements with variables and numbers.

ACKNOWLEDGMENT

Part of this research was conducted as part of the research project "Research on e-learning for next-generation" of the Waseda Research Institute for Science and Engineering, Waseda University. Part of this work was supported by JSPS KAKENHI Grant Numbers JP20K03082, JP19H01721, JP19K04914, and JP17K01101, and Special Account 1010000175806 of the NTT Comprehensive Agreement on Collaborative Research with the Waseda University Research Institute for Science and Engineering. Research leading to this paper was partially supported by the grant as a research working group "information and communication technology (ICT) and Education" of JASMIN.

REFERENCES

[1] R. Robbes and M. Lanza, "A Change-based Approach to Software Evolution," *Electronic Notes in Theoretical Computer Science 166*, pp. 93–109, 2007.

[2] T. Omori and K. Maruyama, "A Method for Extracting Source Code Modifications from Recorded Editing Operations," [in Japanese], *Journal of the Information Processing Society of Japan (IPSJ)*, vol.49, no.7, pp. 2349–2359, 2008.

[3] K. Mori, T. Tanaka, H. Hashiura, A. Hazeyama, and S. Komiya, "Development of an Environment for Gleaning History Information of Programming on a Fine Granularity to Help with Programming Exercise Lesson," *Technical Reports of the IPSJ*, 2013-SE-179, vol.16, pp. 1–6, 2013.

[4] M. Aramoto, M. Kobayashi, M. Nakazawa, M. Nakano, M. Goto, and S. Hirasawa, "Learning Analytics via Visualization System of Edit Record－System Configuration and Implementation," [in Japanese], *78th National Convention of IPSJ*, vol. 4, pp. 527–528, Mar. 2016.

[5] K. Umezawa, M. Aramoto, M. Kobayashi, T. Ishida, M. Nakazawa, and S. Hirasawa, "An Effective Flipped Classroom based on the Log Information of the Self-study," *Proceeding of the 3rd International Conference on Applied Computing & Information Technology (ACIT 2015)*, pp. 263–268, July 2015.

[6] K. Umezawa, M. Kobayashi, T. Ishida, M. Nakazawa, and S. Hirasawa, "Experiment and Evaluation of Effective Grouped Flipped Classroom," *Proceeding of the 5th International Conference on Applied Computing & Information Technology (ACIT 2017)*, pp. 71–76, July 2017.

[7] K. Umezawa, T. Ishida, M. Nakazawa, and S. Hirasawa, "Evaluation by Questionnaire on Grouped Flipped Classroom Method," *Proceeding of the IEEE 10th International Conference on Engineering Education (ICEED2018)*, pp. 87–92, Nov. 2018.

[8] K. Umezawa, T. Ishida, M. Nakazawa, and S. Hirasawa, "Application and Evaluation of a Grouped Flipped Classroom Method," *Proceeding of the IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE2018)*, pp. 39–45, Dec. 2018.

[9] K. Umezawa, M. Nakazawa, M. Goto, and S. Hirasawa, "Development of problem extraction tool for debugging practice using learning history," *Proceeding of the 17th Annual Hawaii International Conference on Education*, pp. 468–470, Jan. 2019.

[10] K. Umezawa, M. Nakazawa, M. Goto, and S. Hirasawa, "Development of Debugging Exercise Extraction System using Learning History," *Proceeding of the 10th International Conference on Technology for Education (T4E 2019)*, pp.244–245, Dec. 2019.

[11] K. Nagashima, S. Cho, H. Manabe, S. Kanemune, and M. Namiki, "Design and Evaluation for Bit Arrow: Web-based Programming Learning Support Environment," [in Japanese] *Proceeding of the IPSJ Technical Report*, pp. 1–8, Feb. 2017.