A Proposal of Mistake Correction Problem for Debugging Study in C Programming Learning Assistant System

Yanhui Jing, Nobuo Funabiki, Soe Thandar Aung, Xiqin Lu, Annisa Anggun Puspitasari, Htoo Htoo Sandi Kyaw, and Wen-Chung Kao

***Abstract*—Currently, *C programming* is taught as the first programming language in many universities around the world due to the easy-to-learn and middle-level nature. However, the confusing concepts of keywords and unfamiliar formality make it difficult for students to study. Therefore, we have previously developed *C programming learning assistance system (CPLAS)* for self-studies of novice students. CPLAS offers several types of exercise problems with the automatic answer marking by *string matching*. In this paper, we propose a *mistake correction problem (MCP)* for *code debugging study* as a new problem type in CPLAS. MCP requests to answer every *mistaken element* and its *correction* in a given corrupt source code. We list up *reserved words* and *common library functions* in *C programming* for candidates of mistaken elements, and implement the *MCP instance generation algorithm*. To help solving MCP instances by a student, we implement the *answer interface* that shows the line number of each mistake, the corrupt code and answer forms in parallel, and the *hint* of suggesting the first character of each answer. For evaluations of the proposal, we generate 20 instances with 91 mistakes for basic grammars, and assign them to 18 university students in Japan, China, and Indonesia. Their answer results confirm the effectiveness of MCP.**

***Index Terms*—C programming, CPLAS, mistake correction problem, automatic generation, answer interface.**

I. Introduction

Currently, *C programming* is taught to undergraduate students in many universities across the world in the first programming courses. *C programming* will be a springboard for learning more advanced and practical programming languages such as *Java* or *Python*. In addition, it is necessary for students in information technology or computer science departments to study how to access memories or registers by computer programs at learning computer architecture [1], [2].

To enhance *C programming* studies, we have developed a web-based *C programming learning assistance system (CPLAS)*. CPLAS offers a variety of programming exercise problems at different levels, including the *grammar-concept understanding problem (GUP)* [3], the *value trace problem (VTP)* [4], the *element fill-in-blank problem (EFP)* [5], the *code completion problem (CCP)* [6], and the *phrase fill-in-blank problem (PFP)* [7]. In any problem type, the answer from a student is instantly marked through *string matching* with the stored correct one at the offered *answer interface* using a web browser [8].

Unfortunately, the current problem types in CPLAS do not cover *code debugging study*, although code debugging should be a central skill for students [9]. Actually, GUP covers *grammar study* in a new programming language by answering the keyword in the given source code that corresponds to each question. VTP covers *code reading study* by tracing important variables or messages in the code. EFP and CCP cover *coding study* partially by filling in the blanks in the code. Therefore, a new type to cover *code debugging study* should be implemented in CPLAS.

In this paper, we propose a *mistake correction problem (MCP)* for *code debugging study* as a new problem type in CPLAS. For a given source code, a MCP instance requests to answer each *mistaken element* and its correction. In the answer interface for MCP, a pair of the input forms for the mistaken element and the correction are prepared for one mistaken element in the corrupt source code, where a student needs to fill in both forms correctly.

To help generating new MCP instances by a teacher, we present the *MCP instance generation algorithm* by modifying the one for GUP instances in [10]. First, *reserved words* and *common library functions* in *C programming* are listed up for candidates of mistaken elements in a code. A total of 67 elements are selected here. Second, these elements are categorized into 11 groups such that the elements having similar roles with each other belong to the same group. Third, the algorithm finds any listed element in the given source code, and replaces it with other randomly selected element in the same group to compose the corrupt source code, assuming that novice students can be often confused among them at programming.

Besides, to help solving MCP instances by a student, we implement the *MCP answer interface* by modifying the one in [8]. This new interface displays the given *corrupt source code* and the *answer input forms* at two columns in parallel. Thus, a student can fill in the forms while seeing the code at the same time. To avoid unnecessary difficulty of the problem, the line number of each *mistaken element* is shown there. Moreover, the first character of each answer will be given as a *hint* when the student needs.

For evaluations of the proposal, we generate 20 MCP instances with 91 mistaken elements using C source codes for basic grammars, and assign them to 18 university students in Japan, China, and Indonesia. Their answer results confirm the effectiveness of MCP.

Manuscript received April 4, 2022; revised July 3, 2022.

Yanhui Jing, N. Funabiki, S. T Aung, X. Lu, and A. A. Puspitasari are with the Department of Information and Communication Systems, Okayama University, Okayama, Japan (e-mail: pf709l29@s.okayama-u.ac.jp, funabiki@okayama-u.ac.jp).

H. H. S. Kyaw is with Division of Advanced Information Technology and Computer Science, Tokyo University of Agriculture and Technology, Tokyo, Japan (e-mail: htoohtoosk@go.tuat.ac.jp).

W.-C. Kao is with Department of Electrical Engineering National Taiwan Normal University, Taipei, Taiwan (e-mail: jungkao@ntnu.edu.tw).

doi: 10.18178/ijiet.2022.12.11.1733

The rest of this paper is organized as follows: Section II discusses related works in literature. Section III presents the mistake correction problem in CPLAS. Section IV shows the implement of the answer interface. Section V evaluates the MCP. Finally, Section VI concludes this paper with future works.

II. Related Works in Literature

In this section, we discuss related works in literature on code debugging studies by novice students.

In [11], Lin *et al*. proposed to observe the cognitive process of a student using an eye tracker at debugging programs to see if and how high- and low-performing students behave differently. Based on the findings, adaptable instructional strategies for students at various performance levels can be devised to improve linked cognitive activities during debugging and foster learning during debugging and programming.

In [12], McCauley *et al*. reviewed literatures related to learning and teaching of debugging computer programs. Debugging is an important skill and has been difficult for novice programmers to learn meanwhile being challenging for computer science educators to teach. They organized four questions: 1) what causes bugs to occur?, 2) what types of bugs occur?, 3) what is the debugging process?, and 4) how can we improve debugging learning and teaching?.

In [13], Luxton-Reilly *et al*. presented an online tool called *Ladebug* that is designed to scaffold the learning of debugging skills. Students follow the structured debugging process to find and fix errors in predefined exercises.

In [14], Lee *et al*. created a generic system architecture and process used in university classes as well as a tool, called the *Virtual Debugging Advisor (ViDA)*, based on the strong relationship observed between common wrong outputs and the corresponding common bugs in students' programs. Their findings are positive, indicating that with *ViDA* enabled, a higher percentage of students were able to modify their own faulty codes, and an overwhelming majority of the respondents deemed *ViDA* to be beneficial to their programming learning.

In [15], Ardimento *et al*. proposed an approach based on reuse of existing bugs of open source systems to provide the informed guidance from the failure sites to the fault positions, and to allow novice programmers to gain debugging experience quickly. The goal is to help novices in reasoning on the most promising paths to follow and conditions to define. They implemented the proposal as a tool named *Debugging Teaching Environment (DTE)* that exploits the knowledge about fault and bug positions.

III. Proposal of Mistake Correction Problem

In this section, we present the mistake correction problem (MCP) and the MCP instance generation algorithm.

A. Definition of Mistake Correction Problem

A MCP instance consists of a corrupt source code that have several mistaken elements and their correct elements with the line numbers in the source code. Any answer from a student is marked through *string matching* with the corresponding correct element.

Fig. 1 shows the answer interface for an example MCP instance. The source code in the left column has three mistakes: **int** at line 3 should be **double** from lines 5 and 7, **scanf** at line 4 be **printf**, and **/lf** at line 5 be **%lf**. In the right column, the corresponding line number is shown to each pair of input forms for the mistaken element and the correct element.

B. Mistaken Element Candidates

To generate a MCP instance automatically, the candidates for mistaken elements are selected from *reserved words* and *common library functions* in *C programming* that often appear in source codes and should be studied by novice students. Actually, a total of 67 elements are selected here. Then, these candidates are categorized into 11 groups such that the ones having similar roles with each other belong to the same group. Table I shows the selected elements with 11 groups. For a given source code, each candidate element will be replaced by randomly selected another element in the same group, since novice students may be confused among the elements in the same group at writing a source code.

C. MCP Generation Algorithm

A MCP instance is generated through the following procedure:

1) Read a *C programming* source code file.
2) Select a mistaken element candidate in Table I that appears in the source code if all the following conditions are satisfied:
   2-1) The same candidate is not selected before.
   2-2) Any other element at the same line is not selected before.
   2-3) The generated 0-1 random number for the selection is smaller than 0.5.
3) Generate the corresponding mistaken element to this selected candidate.
   3-1) Choose another candidate in the same group as the mistaken element randomly.
   3-2) Replace the selected candidate by it in the source code to make the *corrupt source code*.
   3-3) Keep the mistaken element, the original one, and the line number in the source code for the *correct answers*.
4) Combine the *corrupt source code* and *correct answers* into one text file.
5) Run the *answer interface generator* with the text file to make the *HTML*/*CSS*/*JavaScript* files for the MCP instance.

D. MCP Instance Generation Example

Code 1 shows the original source code for this example. This code requests to get two double-type values from the standard input and outputs their multiplication in the standard output.

Code 2 shows the *corrupt source code* generated by the algorithm. From the original source code, three candidates in Table I, namely, **double** at line 3, **printf** at line 4, and **%** at line 5, are selected. Then, **int**, **scanf**, and **/**, are selected from the same groups for their corresponding mistaken elements.

TABLE I: MISTAKEN ELEMENT CANDIDATES FOR MCP

group	mistaken element candidates
1	int short long float double char
2	while if for switch case
3	+ - * / % & ++ –
4	== != ><>= <=
5	%d %ld %lf %c %s
6	printf scanf
7	stdio stdlib ctype string math time assert signal stddef
8	getchar putchar gets puts
9	strlen strcpy strncpy strcmp fopen fclose exit fputc fgetc fscanf fputs fgets
10	malloc calloc
11	([{ }])

Code 1: Original source code example.

```
1: # include < stdio.h >
2: int main() {
3:   double a, b, product;
4:   printf("Enter two numbers:");
5:   scanf("%lf%lf", &a, &b);
6:   product = a*b;
7:   printf("Product = %.2lf", product);
8:   return 0;
9: }
```

Code 2: Corrupt source code example.

```
1: # include < stdio.h >
2: int main(){
3:   int a, b, product;
4:   scanf("Enter two numbers:");
5:   scanf("/lf%lf", &a, &b);
6:   product = a*b;
7:   printf("Product = %.2 lf", product);
8:   return 0;
9: }
```

IV. MCP ANSWER INTERFACE

In this section, we present the MCP answer interface by modifying the existing interface in [8].

This new interface shows the given corrupt source code and the *answer input forms* at two columns in parallel. Thus, a student can fill in the input forms while seeing the code at the same time. To avoid unnecessary difficulty of the problem, the line number of each mistaken element is shown there. Moreover, the first character of each answer will be given as a *hint* when the student needs help.

A. Two-Column Layout

The interface or the web page in [8] has the one-column layout. In this interface, the *corrupt source code* appears first, and then, the *answer input forms* appear. Thus, a student has to scroll the page many times to see the code and input the forms.

On the other hand, the new interface has the two-column layout as shown in Fig. 1. In this interface, the *corrupt source code* and the *answer input forms* appear in parallel.

Thus, a student does not need to scroll the page to see the code and input the forms, unless the code is very long or a lot of mistaken elements are included. To deal with a long code, the designated scroll bar is prepared to scroll the *corrupt source code*.

B. Answering Time and Count Display

To encourage a student to solve the given MCP instances in a short time with the less number of answer submissions, the MCP answer interface shows the *elapsed time* since the student started to solve the current instance and the number of answer submission times by clicking the ”Answer” button, as shown in Fig. 2. It indicates that 9*min*. 52*sec*. have passed since the student started solving the instance and clicked the button 8 times.

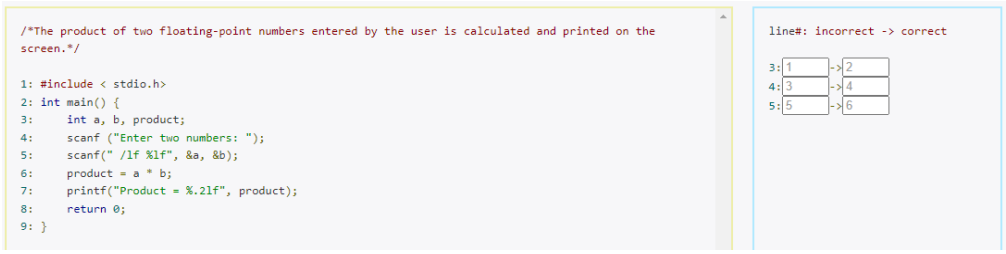

Fig. 1. MCP answer interface.

Fig. 2. Answering time and count example.

C. Hint Button

To avoid giving up solving an MCP instance by a student, the first character of each correct element in the instance will appear as the *hints*, when he/she clicks the ”Hint” button, as shown in Fig. 3. The reason of showing the first character is that this hint can be easily generated from the stored correct element in the text file.

However, to prevent a student from using the hint button facilely, the hint will appear only when the answering time exceeds 10 minutes and the number of answers exceeds 5 times.

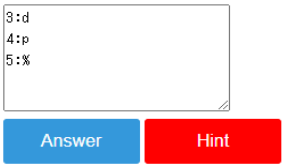

Fig. 3. Hint button.

TABLE II: GENERATE MCP INSTANCES

ID	topic	# of lines	# of mistakes
1	95.83	3.22	4
2	88.89	3.28	8

3	94.44	3.17	6
4	90.28	2.56	4
5	95.56	4.06	10
6	89.58	3.11	8
7	88.19	7.00	8
8	94.44	1.83	6
9	97.22	2.22	10
10	93.25	5.39	14
11	95.83	2.17	8
12	69.50	8.33	12
13	90.74	2.94	6
14	97.22	2.28	10
15	88.89	6.27	14
16	86.67	6.06	10
17	88.10	4.83	14
18	84.26	6.94	12
19	88.89	4.94	8
20	88.61	9.50	8
average	90.32	4.51	9
SD	6.04	2.17	3

V. Evaluation

In this section, we evaluate the proposed MCP in CPLAS through applications to 18 university students in Japan, China, and Indonesia.

A. Generate MCP Instances

For evaluations, we generate 20 MCP instances with the total of 91 mistaken elements using 20 source codes for basic *C programming* grammar topics. Table II shows the topic, the number of lines in the source code, and the number of mistaken elements in each instance.

B. Solution Results for Individual MCP Instances

First, we analyze the solution results for the individual MCP instances by all the students. Fig. 4 shows the average correct answer rate and the average number of answer submission times for each MCP instance. The average and standard deviation (SD) among all the instances are 90.32% and 6.04% for the correct rate, and are 4.51 and 2.17 for submission times, respectively. These results suggest that the generated MCP instances have medium difficulty for *C programming* beginners.

C. Hard MCP Instances

Next, we analyze hard MCP instances for novice students. Fig. 4 indicates that the instance at ID=12 exhibits the lowest correct rate and the second highest number of submissions. Firstly, as shown in Code 3, it contains a mistake on the **long long** data type, which is not common to novice students. It is used to deal with large integer values by storing 64 bits of data, which takes twice as much memory as the **long** data type. To print or scan the **long long** data type, the prefix become **ll**. Thus, **%lld** is used here. Secondly, at line 11, a mistaken element **}** is included. Some students do not notice the distinction between **}** and **)**. Finally, some students are confused at the conditional statements of == and **!=** in this instance.

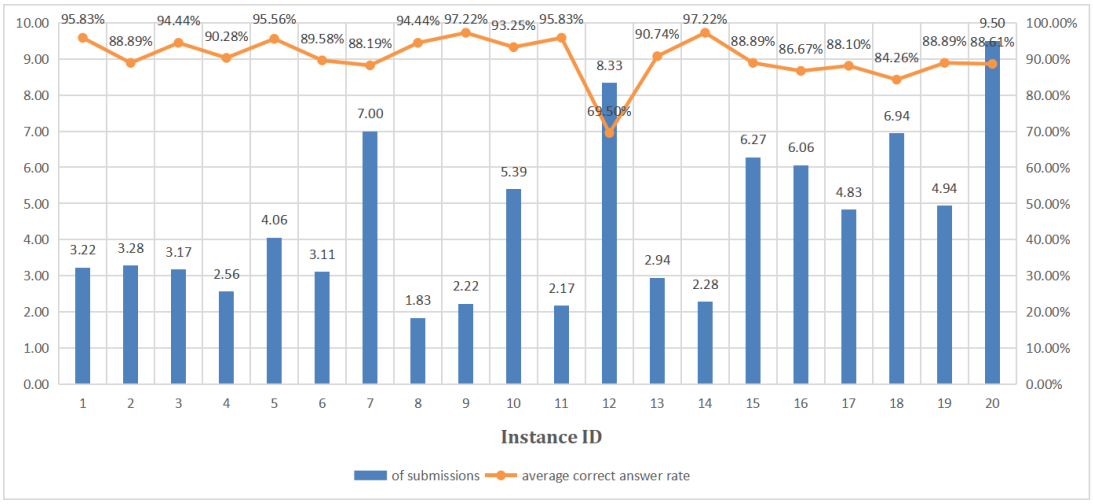

Fig. 4. Solution results for individual MCP instances.

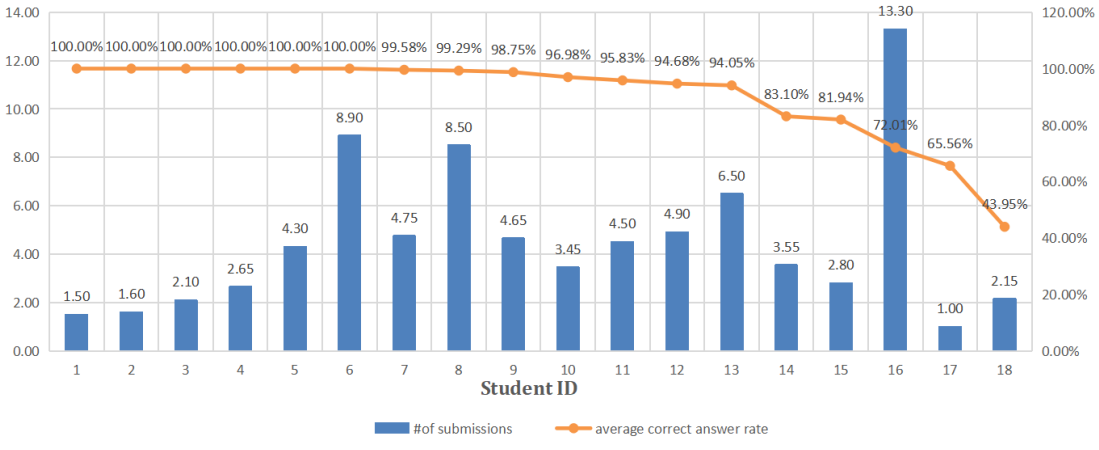

Fig. 5. Solution results for individual students.

Fig. 4 also indicates that the instance at ID=20 exhibits the highest number of submissions. As in Code 4, it contains mistakes on **strlen** and **strcpy** regarding the file access, which may not be well understood by novice students. Therefore, we will add brief explanations on **long long**, **==**, and **strcpy** in the corresponding MCP instances, to easily remind students of them, which will be in future works.

D. Solution Results for Individual Students

Third, we analyze the solution results for the individual students in all the MCP instances. Fig. 5 shows the average correct answer rate and the average number of answer submission times for each student. Their SD among all the students are 15.14% and 3.04, respectively. Six students among 18 achieved the 100% correct rate, and 13 students did higher than 94%. Only one student did not reach 50%. The results suggest that most of the students seriously solved the MCP instances.

VI. Conclusion

This paper proposed the mistake correction problem (MCP) for code debugging study in C programming learning assistance system (CPLAS). It requests to answer every mistaken element and its correction in the given corrupt source code. To help generating MCP instances from source codes, 67 candidate mistaken elements with 11 groups were selected from reserved words and common library functions in C programming, and the MCP instance generation algorithm was implemented. The answer interface for solving MCP instances was also presented, which shows the corrupt code and the input forms in parallel and offers the hints when requested. For evaluation, 20 instances with 91 mistakes were generated using source codes for C programming basic grammars, and were assigned to 18 university students in Japan, China, and Indonesia. Their answer results confirmed the effectiveness of the proposal. In future works, we will add brief explanations of hard programming concepts in the corresponding MCP instances, generate new MCP instances on other topics that students may not understand well, and assign them in C programming courses.

Code 3: Mistaken code at ID=12.

```
1:#include math.h
2:#include stdio.h
3:int convert(long double bin);
4:int main() {
5:    long long bin;
6:    printf("Enter a binary number: ");
7:    scanf("%lld", &bin);
 8:   printf("%lld in binary = %d in octal", bin, convert(bin));
9:   return 0;
10: }
11:int convert(long long bin} {
12:   float oct = 0, dec = 0, i = 0;
13:   while (bin != 0) {
14:       dec += (bin % 10) * pow(2, i);
15:       ++i;
16:   bin /= 10;
17:    }
18:    i = 1;
19:    if (dec != 0) {
20:        oct += (dec % 8) * i;
21:        dec /= 8;
22:        i *= 10;
23:    }
24:    return oct;
25:    }
```

Code 4: Mistaken code at ID=20.

```
1:#include stdio.h
2:#include string.h
3:int main() {
4:   char str[5][50], temp[50];
5:   scanf ("Enter 5 words: ");
6:   for (int i = 0; i < 5; --i) {
7:      fgets(str[i], sizeof(str[i]), stdin);
8:   }
9:   for (int i = 0; i != 5; ++i) {
10:      for (int j = i + 1; j < 5; ++j) {
11:         if (strcmp(str[i], str[j]) > 0) {
12:            strlen(temp, str[i]);
13:            strcpy(str[i], str[j]);
14:            strcpy(str[j], temp);
15:         }
16:      }
17:   }
18:   printf("\nIn the lexicographical order: \n");
19:   for (int i = 0; i < 5; ++i) {
20:      fputs(str[i], stdout);
21:   }
22:   return 0;
23:    }
```

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Y. Jing conducted the research and wrote the paper. N. Funabiki and W.-C. Kao reviewed and finalized the paper. S.

T. Aung, X. Lu, and A. A. Puspitasari collected and analyzed the data. H. H. S. Kyaw collected the source codes. All the authors had approved the final version.

ACKNOWLEDGMENT

We would like to thank the students to answer the MCP instances and give us valuable comments. They are inevitable to complete this paper.

REFERENCES

[1] K. Gondow and Y. Arahori, "Why do we need the C language in programming courses?" in *Proc. ICSOFT*, pp. 549-556, 2018.
[2] (Sep. 2021). Why learning C Programming is a must? Geeks-forGeeks. [Online]. Available: https://www.geeksforgeeks.org/why-learning-c-programming-is-a-must/
[3] X. Lu, S. T. Aung, H. H. S. Kyaw, N. Funabiki, S. L. Aung, and T. T. Soe, "A study of grammar-concept understanding problem for C programming learning," in *Proc. LifeTech*, pp. 162-165, March 2021.
[4] X. Lu, N. Funabiki, H. H. S. Kyaw, E. E. Htet, S. L. Aung, and N. K. Dim, "Value trace problems for code reading study in C programming," *Adv. Sci. Tech. Eng. Syst. J. (ASTESJ)*, vol. 7, no. 1, pp. 14-26, Jan. 2022.
[5] H. H. S. Kyaw, N. Funabiki, S. L. Aung, N. K. Dim, and W.-C. Kao, "A study of element fill-in-blank problems for C programming learning assistant system," *Int. J. Inform. Edu. Tech. (IJIET)*, vol. 11, no. 6, pp. 255-261, June 2021.
[6] H. H. S. Kyaw, E. E. Htet, N. Funabiki, M. Kuribayashi, T. Myint, P. P. Tar, N. W. Min, H. A. Thant, and P. H. Wai, "A code completion problem in C programming learning assistant system," in *Proc. ICIET*, pp. 34-40, March 2021.
[7] X. Lu, N. Funabiki, H. H. S. Kyaw, E. E. Htet, S. L. Aung, and N. K. Dim, "Value trace problems for code reading study in C programming," *Adv. Sci. Tech. Eng. Syst. J. (ASTESJ)*, vol. 7, no. 1, pp. 14-26, Jan. 2022.
[8] N. Funabiki, H. Masaoka, N. Ishihara, I-W. Lai, and W.-C. Kao, "Offline answering function for fill-in-blank problems in Java programming learning assistant system," in *Proc. ICCE-TW*, pp. 324-325, May 2016.
[9] T. Michaeli and R. Romeike, "Improving debugging skills in the classroom: the Effects of teaching a systematic debugging process," in *Proc. WiPSCE*, pp. 1–7, Oct. 2019.
[10] S. T. Aung, N. Funabiki, Y. W. Syaifudin, H. H. S. Kyaw, S. L. Aung, N. K. Dim, and W.-C. Kao, "A proposal of grammar-concept understanding problem in Java programming learning assistant system," *J. Adv. Inform. Tech. (JAIT)*, vol. 12, no. 4, Oct. 2021.
[11] Y. T. Lin, C. C. Wu, T. Y. Hou, Y. C. Lin, F. Y. Yang, and C. H. Chang, "Tracking students' cognitive processes during program debugging — An eye-movement approach,", vol. 59, no. 3, pp. 175-186, Aug. 2016.
[12] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: A review of the literature from an educational perspective," *Comput. Sci. Educ.*, vol. 18, no. 2, pp. 67–92, June 2008.
[13] A. Luxton-Reilly, E. McMillan, E. Stevenson, E. Tempero, and P. Denny, "Ladebug: An online tool to help novice programmers improve their debugging skills," in *Proc. ITiCSE*, pp. 159–164, 2018.
[14] V. C. S. Lee, Y. T. Yu, C. M. Tang, T. L. Wong, and C. K. Poon, "ViDA: A virtual debugging advisor for supporting learning in computer programming courses," *J. Comput. Assist. Learn. (JCAL)*, vol. 34, no. 3, pp. 243-258, June 2018.
[15] P. Ardimento, M. L. Bernardi, M. Cimitile, and G. D. Ruvo, "Reusing bugged source code to support novice programmers in debugging tasks," *ACM Trans. Comput. Edu.*, vol. 20, no. 1, pp. 1–24, March 2020.

Y. Jing received the B.S. degree in information management and information system (Japanese-English bilingual strengthening) from Dalian University of Foreign Languages, China, in 2020. She is currently a master student in electronic information systems at Okayama University, Japan. Her research interests include educational technology.

Nobuo Funabiki received the B.S. and Ph.D. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1984 and 1993, respectively. He received the M.S. degree in electrical engineering from Case Western Reserve University, USA, in 1991. From 1984 to 1994, he was with Sumitomo Metal Industries, Ltd., Japan. In 1994, he joined the Department of Information and Computer Sciences at Osaka University, Japan, as an assistant professor, and became an associate professor in 1995. In 2001, he moved to the Department of Communication Network Engineering at Okayama University as a professor. His research interests include computer networks, optimization algorithms, educational technology, and Web technology. He is a member of IEEE, IEICE, and IPSJ.

S. T. Aung received the B.E. degree in information technology from Thanlyin Technological University, Myanmar, in 2017. She is currently a master student at Okayama University, Japan. Her research interests include educational technology.

X. Lu received the B.S. degree in electronic information engineering from Hubei University of Economics, China, in 2017, and received the M.S degree in electronic information systems from Okayama University, Japan, in 2021, respectively. She is currently a Ph.D. student at Okayama University, Japan. She received the OU fellowship in 2021. Her research interests include educational technology.

A. A. Puspitasari received the B.E. degree in telecommunication engineering from Politeknik Elektronika Negeri Surabaya (PENS), Indonesia, in 2021. She is currently an adjunct researcher at Okayama University, Japan. Her research interests include educational technology and wireless communication systems.

H. H. S. Kyaw received the B. E. and M. E. degrees in information science and technology from University of Technology (Yatanarpon Cyber City), Myamar, in 2015 and 2018, respectively. She received Ph.D. from Graduate School of Natural Science and Technology at Okayama University, Japan in 2021. She is currently an assistant professor from Tokyo University of Agriculture and Technology, Japan. Her research interests include educational technology and web application systems.

W.-C. Kao received the M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taiwan, in 1992 and 1996, respectively. Since 2004, he has been with National Taiwan Normal University (NTNU), Taipei, Taiwan, where he is currently the Research Chair Professor at Department of Electrical Engineering and the Dean of College of Technology and Engineering. His current research interests include system-on-a-chip (SoC) as well as embedded software design, flexible electrophoretic display, and machine vision system.