An Educational Support System based on Automatic Impasse Detection in Programming Exercises

Koichi YAMASHITA[a*], Takumi SUGIYAMA[b], Satoru KOGURE[c], Yasuhiro NOGUCHI[c], Tatsuhiro KONISHI[c] & Yukihiro ITOH[c]
[a]*Faculty of Business Administration, Tokoha University, Japan*
[b]*Graduate school of Integrated Science and Technology, Shizuoka University, Japan*
[c]*Faculty of Informatics, Shizuoka University, Japan*
*yamasita@hm.tokoha-u.ac.jp

Abstract: In this paper, we develop an educational support system based on an automatic impasse detection method. In programming education, novice learners occasionally experience various impasses during coding exercises. A learner's impasse log, which includes information such as the type of the impasses and timestamps at which they reached the impasses, would be useful for the learning and teaching of programming, such as for reflective learning and one-on-one instructions. Although several systems have been developed to understand students' programming statuses, there is no system based on the students' impasse in programming. We developed a ruleset to detect impasses automatically based on the indications of students' codes and errors and improved the detection ability through an experimental introduction in actual classrooms. Using this framework, we developed a system that supports learners to perform reflective learning. We conducted a pilot experiment to evaluate the contribution of our system to reflective learning and obtained positive evaluation results.

Keywords: Educational support system, automatic impasse detection, reflective learning

1. Introduction

In programming education, novice learners occasionally experience various impasses during coding exercises; they might be unable to identify correct implementations or resolve certain compilation errors for a long time, and therefore not be able to proceed with their exercise. However, in the typical style of programming exercises, teachers and teaching assistants tend to have a relatively large number of students to monitor and understand every student's impasses individually and precisely.

We consider that, unlike students coding effortlessly, those reaching an impasse would have some indication during their coding process. If we can identify such indications of impasses by observing how novice learners perform their coding exercises and how they reach impasses, it could be feasible to generate a ruleset for impasse detection. Based on this consideration, it would be possible to implement an automatic, rule-based impasse detection and record it in an impasse log.

A learner's impasse log, which includes the type of the impasse, a snapshot of the program-code, and the compilation and run-time errors at that time, can be useful for learning and teaching programming; specifically, we can exemplify student's reflective learning, one-on-one instruction during the exercise, and classroom designing. During reflective learning, students can reflect how they had reached impasse and how they had broken it using their personal impasse log as a clue. During classroom exercise, teachers can locate the students reaching an impasse and identify the type of impasse by monitoring their impasse logs in real-time. For classroom designing, the impasse logs can focus the teachers' attention on learning items that had been difficult for many students to understand.

In this paper, we describe a system that records impasse logs and visualizes the summary. To develop the ruleset for impasse detection, we observed some novice learners' coding activities and whether the learners reached an impasse. Furthermore, we recorded the snapshot of their program-code and the compilation and run-time errors every a certain time period. Based on the codes and errors recorded at the time when the learners had reached an impasse, we generated a ruleset for impasse detection. We describe the ruleset development in Section 3 and our system in Section 4. We introduced our system into an actual class of 110 students for three months, and collected nearly 2000 impasse logs. In Section 5, we describe the pilot experiment that we conducted to verify our system's contribution to reflective learning. The evaluation results suggest that our system supports student's reflection to a certain degree.

2. Related Work

The concept to support learning and education by monitoring learners' behaviors has a long history in computing education research. The most intuitive approach is tracking learners' activities using the logging data of course management systems (Mazza & Dimitrova, 2004) in a certain computer-based learning environment (Biswas & Sulcer, 2010). In the context of programming education, Thomas et al. (2003) monitored the generic computer usage of programming students, recording low level actions such as mouse clicks, typing, and window changes. They identified the abstracted meanings and purposes of students' action by analyzing the recorded actions. However, additional research is required because it is very difficult to analyze large-scale data using a bottom-up approach and to extract meaningful actions from them.

Compiler error messages are frequently used to understand the status of students' programming. Brown and Altadmri (2014) collected the error messages of a large number of students (more than 100,000) from numerous institutions and analyzed the frequently made programming mistakes of novice learners. They highlighted teachers' subjective impressions about novice learners' common mistakes. Hartmann, MacDougall, Brandt, and Klemmer (2010) developed the HelpMeOut system that supports learners to debug compilation errors. HelpMeOut tracks code evolution over time and collects learners' modifications that take program-code from an error state to error-free state. When a learner experiences a compilation error, HelpMeOut, as a sample solution, suggests the modification of the other learner who had experienced the same compilation error in the past.

Piech et al. (2010) adopted an approach that uses the changes in learners' program-code. To model how students learn programming, they collected students' code at certain time intervals and at every time a student compiled a project. They developed a programming model of students by modeling student progress, based on a Hidden Markov Model. They estimated transition parameters using recoded codes as observed outputs. Their model could potentially provide insights into whether students require interventions.

Our study is novel because we supported learners to not proceed with their programming exercises, but to perform reflective learning. Our basic idea also includes supporting teachers to provide one-to-one instruction in the exercise, and to design classroom. On that point, our approach is similar to the study of Piech et al. However, our approach is different from it in that ours uses both learners' codes and errors and is based on an exercise-independent ruleset. Moreover, our impasse detection is based on abstracted indications of learners' coding activities, whereas that of Piech et al. (2010) is based on the superficial indications appearing on the learners' codes.

3. Ruleset Development for Impasse Detection

3.1 Ruleset Development Based on Observing Learners' Coding Activities

For impasse detection, we recognized the indications appearing only on learner's program-code and compilation/run-time errors. Learners' coding activities provide other information such as the points of their gaze or the motions of their eyes, captured using eye tracking devices. However, this information is not realistic because such devices might not be introduced into actual coding exercises normally, and hence they would impose a burden to the learners. An externalization of learner's thinking is also difficult for a similar reason. Therefore, we developed a subsystem to record four types of learner information: every one-minute program-code, every compiled program-code, every (if any) compilation error, and every (if any) run-time error.

To collect instances of impasse indications appearing on these types of information, we invited five voluntary undergraduate students majoring in computer science. We gave them programming tasks and observed their coding activities. The task involved solving few problems from a past regional Association for Computing Machinery / International Collegiate Programming Contest (ACM-ICPC) that the students had never seen. The total number of observed students was 15 and the total time of observation was approximately 30 hours. During their problem-solving, we collected abovementioned information using our subsystem and observed whether the learners reached an impasse; if yes, what were the causes of the impasse.

Through the observations, we classified learners' impasses into the following four types:

Type 1. Learners unable to develop implementation strategy (including requirements interpretations, algorithm and data-structures design, and program design).

Type 2. Learners unable to resolve compilation errors.

Type 3. Learners unable to resolve run-time errors.

Type 4. Learners unable to modify the code to function as they expected.

We investigated all recorded codes and errors on each impasse type. Table 1 presents the indications appearing on the codes and errors for each impasse type derived from the investigation.

Table 1: Indications appearing on the codes and errors for each impasse type.

Type of impasse	Indications
Type 1	(1-1) The learner has not modified the code for a long time.
Type 2	(2-1) Compilation errors do not decrease even when the learner has compiled successively.
	(2-2) Compilation errors once resolved have reoccurred.
Type 3	(3-1) Same run-time errors have occurred successively.
Type 4	(4-1) The learner has repeatedly modified the same position in the code.
	(4-2) Statement calling a standard output function, such as printf(), with some variable as arguments has been added and then removed.
	(4-3) Compilation has been repeatedly executed in a short time period.
	(4-4) Code has been re-modified to a previous code.
	(4-5) A large part of code has been modified at once.
	(4-6) Statements including the same variable have been modified successively.

The automatic impasse detection could be performed by recording learners' codes and errors in real time, scanning the indications (Table 1) in the recorded materials, and reporting corresponding type of impasse if an indication was found. Based on results of this investigation, we generated the ruleset for impasse detection, presented in Table 2, setting underlined threshold parameters. The following threshold parameters were derived from our investigation.

Table 2: Ruleset for automatic impasse detection.

Type of impasse	Conditions to detect
Type 1	(1-1) The learner has not modified the code for ≥15 minutes.
Type 2	(2-1) Compilation errors have not decreased between any two sequential compilations by the learner.
	(2-2) Compilation errors once resolved have reoccurred.
Type 3	(3-1) Same run-time errors occurred ≥7 times, successively.
Type 4	(4-1) Same position in the code has been successively modified by the learner ≥2 times.
	(4-2) Statement calling a standard output function, such as printf(), with some variable as arguments has been added and then removed.
	(4-3) Compilation has been executed ≥2 times in ≤3 minutes.
	(4-4) Code has been re-modified to a previous code.
	(4-5) ≥15% of the code has been modified at once.
	(4-6) Statements including the same variable have been modified ≥3 times, successively.

3.2 Improving the Detection Ability of the Ruleset

We introduced the subsystem of impasse detection into actual classes of an introductory Java programming course, from October 7, 2016, to November 17, 2016, (6 weeks). We aimed to evaluate the detection ability of the ruleset described in the previous subsection, and to adjust rules or threshold parameters to improve the detection ability. The course named "Programming" is offered to first grade undergraduate students majoring in computer science. A total of 110 students were enrolled in the course. The subsystem detected 2104 impasses within our experimental introduction, and recorded them into impasse logs, which included the impasse types and the snapshot of the codes and errors. To verify the validity of the ruleset, we measured the recall and precision of detected impasse logs as follows:

$$precision = \frac{N\ of\ correct\ detections}{N\ of\ correct\ detections + N\ of\ incorrect\ detections}$$

$$recall = \frac{N\ of\ correct\ detections}{N\ of\ impasses\ detected\ by\ hand\ from\ all\ recorded\ codes\ and\ errors}$$

In the precision calculation, each impasse of all 2104 detections was manually classified into correct, incorrect, or non-identifiable, judging from the codes and errors recorded with the impasse. As the result, we obtained 938 correct, 1004 incorrect, and 162 non-identifiable detections; hence, the precision was .483. In the recall calculation, the number of all recorded codes and errors was considerably large to classify each of them into detect or not detect. Therefore, a 5% random sample of all records was classified. Consequently, we obtained 108 impasses to detect and 91 correct detections by the ruleset from the sample; hence, the recall was .843.

We reviewed the logs of misdetected impasses and found overdetections frequently. Moreover, we found misdetection because at condition (2-1) a trivial syntax error occurred while a student was resolving another compilation error; hence, the total number of errors did not decrease. To improve the detection ability, we adjusted the threshold parameters of the ruleset and modified condition (2-1) in Table 2. Table 3 provides our modified ruleset for impasse detection; the revisions are underlined.

Table 3: Modified ruleset for automatic impasse detection.

Type of impasse	Conditions to detect
Type 1	(1-1) The learner has not modified the code for ≥15 minutes.
Type 2	(2-1) The learner has not resolved identical compilation errors during four sequential compilations. (2-2) Compilation errors once resolved have reoccurred.
Type 3	(3-1) Same run-time errors have occurred ≥7 times, successively.
Type 4	(4-1) Same position in the code has been modified by the learner ≥4 times, successively. (4-2) Statement calling a standard output function, such as printf(), with some variable as arguments has been added and then removed. (4-3) Compilation has been executed ≥5 times in ≤2 minutes. (4-4) Code has been re-modified to a previous code. (4-5) ≥25% of the code has been modified at once. (4-6) Statements including the same variable have been modified ≥5 times, successively.

With the modified ruleset, 1836 impasses were detected from the same codes and errors. Of the total detections, 922 were correct, 756 were incorrect, and 158 were non-identifiable; hence, the precision was .549. We obtained 89 correct detections by the modified ruleset from the same sample containing 108 impasses to detect; hence, the recall was .824. *F*-measure, the harmonic mean of recall and precision, improved from .614 for the initial ruleset to .659 for the modified ruleset.

4. Overview of Our System Architecture

We developed the system visualizing the impasse summary, incorporating the subsystem for automatic impasse detection mentioned in the previous section. Java was the target programming language. Figure 1 displays our system architecture.

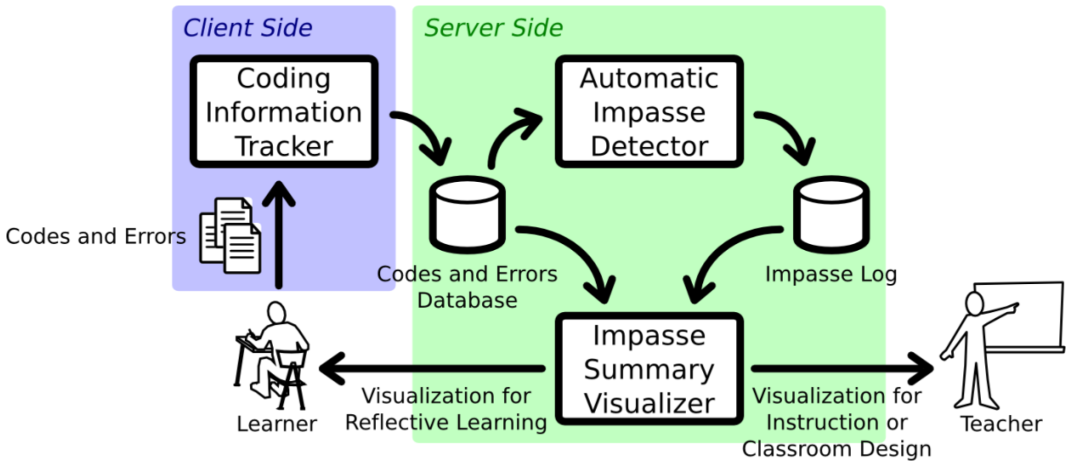
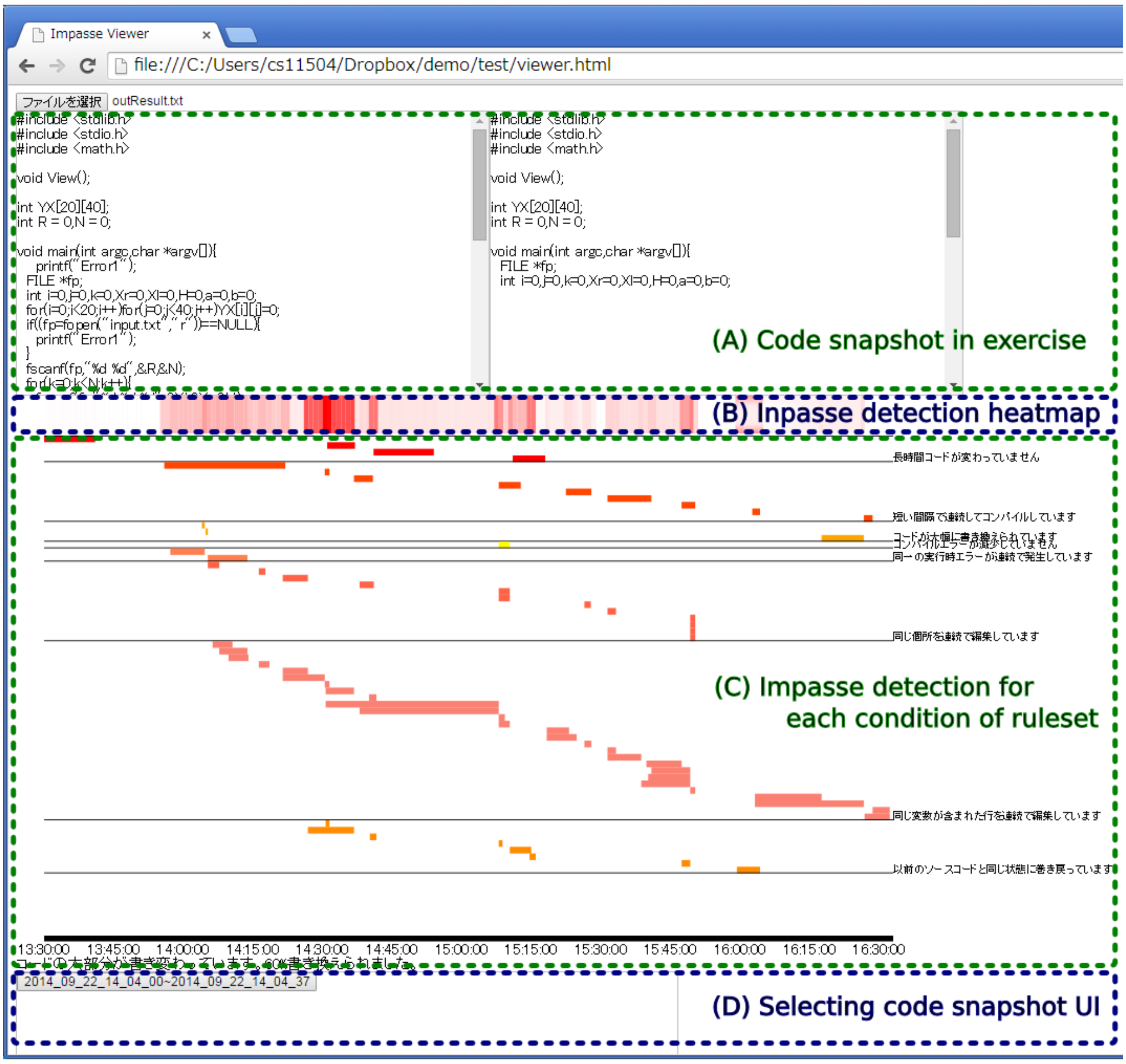

Client Side
Server Side
Coding Information Tracker
Automatic Impasse Detector
Codes and Errors
Codes and Errors Database
Impasse Log
Impasse Summary Visualizer
Learner
Visualization for Reflective Learning
Visualization for Instruction or Classroom Design
Teacher

Figure 1. Overview of our system architecture.

Our system consists of the coding information tracker, the automatic impasse detector, and the impasse summary visualizer. The coding information tracker records for each learner every one-minute program-code that the learner is coding, every compiled program-code, every (if any) compilation error, and every (if any) run-time error. The automatic impasse detector scans the coding information in real-time, detects each learner's impasse based on the ruleset described in the previous section, and records the type of impasse with coding information as the impasse log. The impasse summary visualizer summarizes the recorded impasse log and visualizes based on the user's demand, namely, for reflective learning, one-on-one instruction during the exercise, or designing classroom. We have described the implementation of the coding information tracker and the automatic impasse detector as the subsystem in the previous section. Up to the present time, the impasse summary visualizer was also implemented only for learners' reflective learning.

In reflective learning, learners reflected on the status of their understanding and identified the learning target that they had failed to understand. Hence, the impasse summary visualizer for reflective learning was required to visualize an overview of the types of impasses the learners had reached in exercises, and the snapshots of program-codes when they were experiencing impasses. Figure 2 provides the visualization of our impasse summary visualizer, including the snapshots of codes in (A), heat-map style summary of impasse detection in (B), impasse detection for each condition of the ruleset in (C), and interface of selecting a snapshot of the learners' code in (D).

If all of the detected impasses were listed for the reflective learners, they would not understand what they should begin reflecting on. The visualizer provides the time-based summary of detected impasses in heat map style in (B), and each time slot is colored more deeply according to the more number of satisfied impasse conditions. Moreover, each satisfied condition is visualized on the time base in (C), presenting a period of detecting time. According to the visualizations in (B) and (C), learners clicked on a condition of impasse to reflect in (C), and then click on the system visualize buttons in (D) of the time slot included in the time period to detect the corresponding condition. When the learners clicked on a time slot button in (D), the system visualized the source codes at the beginning and end of the selected time slot.


Impasse Viewer
file:///C:/Users/cs11504/Dropbox/demo/test/viewer.html
ファイルを選択
outResult.txt
(A) Code snapshot in exercise
(B) Inpasse detection heatmap
(C) Impasse detection for each condition of ruleset
(D) Selecting code snapshot UI
長時間コードが変わっていません
短い間隔で連続してコンパイルしています
コードが大幅に書き換えられています
コンパイルエラーが減少していません
同一の実行時エラーが連続で発生しています
同じ個所を連続で編集しています
同じ変数が含まれた行を連続で編集しています
以前のソースコードと同じ状態に巻き戻っています
13:30:00 13:45:00 14:00:00 14:15:00 14:30:00 14:45:00 15:00:00 15:15:00 15:30:00 15:45:00 16:00:00 16:15:00 16:30:00
2014_09_22_14_04_00~2014_09_22_14_04_37

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void View();

int YX[20][40];
int R = 0,N = 0;

void main(int argc,char *argv[]){
    printf("Error1");
  FILE *fp;
  int i=0,j=0,k=0,Xr=0,Xl=0,H=0,a=0,b=0;
  for(i=0;i<20;i++)for(j=0;j<40;j++)YX[i][j]=0;
  if((fp=fopen("input.txt","r"))==NULL){
    printf("Error1");
  }
  fscanf(fp,"%d %d",&R,&N);
  for(k=0;k<N;k++){
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void View();

int YX[20][40];
int R = 0,N = 0;

void main(int argc,char *argv[]){
  FILE *fp;
  int i=0,j=0,k=0,Xr=0,Xl=0,H=0,a=0,b=0;
```

コードの大部分が書き変わっています。60%書き換えられました。

Figure 2. Visualization of our impasse summary visualizer.

5. Pilot Experiment

We conducted a pilot experiment to evaluate the contribution of our system to reflective learning. The experimental hypotheses are summarized as follows:

H1: Our system allows learners to easily perform reflective learning.
H2: The impasses detected and visualized by our system is useful for reflective learning.
H3: Our system promotes deeper reflection in reflective learning.

The subjects were five first grade undergraduate students majoring in computer science, who had participated in classrooms that had introduced our subsystem, experimentally described in Section 3. The targets of reflective learning were the actual exercises in the classrooms.

First, we explained the meaning and purpose of this experimental reflective learning and how to externalize the achievement of the reflection to the subjects in five minutes. Thereafter, we had the subjects perform the following three stages of reflective learning:

Reflection 1: Reflection depending on their memory and exercise materials only.

Reflection 2: Reflection with their impasse log recorded by our subsystem's tracker and detector.

Reflection 3: Reflection using our system that includes a tracker, detector, and visualizer.

The subjects provided written briefs about their learning achievements at the end of every stage. After all of the reflective learning, we administered a questionnaire survey related to the subjects' activities of reflective learning and the contribution of our system to it. The questionnaire contained the following five items on a five-point scale and a comment item:

Q1. How easy was it to perform reflective learning in Reflection 1?

Q2. How easy was it to perform reflective learning in Reflection 2?

Q3. How easy was it to perform reflective learning in Reflection 3?

Q4. How much did the impasses detected by our system contribute to reflective learning?

Q5. How instinctively correct were the impasses detected by our system?

Table 4 provides the responses of each subject for each questionnaire item. For each item, a higher score indicated a more positive response. The responses to Q1, Q2, and Q3 suggest that the subjects preferred reflection with impasse log to reflection without any support of our system, and preferred reflection with system's visualization to reflection with raw log data. In addition, subjects #3 and #5 were exceptions; subject #3 preferred reflecting with raw impasse log because his coding time was extremely shorter than all other subjects and his impasse log was so small that he could easily observe his entire codes and errors without any summarization. However, subject #5 stated the most positive points for all reflections in his responses because his coding time was extremely long. He experienced impasses so frequently that all reflections sufficiently supported his understandings. On the other hand, his response in the comment item was that the easiest reflective learning was in Reflection 3. Based on this discussion, we consider that these results support H1. Moreover, the responses to Q4 and Q5 indicate that the subjects positively assessed the validity of impasse detections and the contributions of the detected impasses. We consider that these results support H2.

Table 4: Questionnaire responses of each subject.

Item	Subj #1	Subj #2	Subj #3	Subj #4	Subj #5	Ave.
Q1	3	2	1	4	5	3.00
Q2	3	2	5	2	5	3.40
Q3	5	4	4	5	5	4.60
Q4	5	4	4	5	5	4.60
Q5	4	4	5	4	5	4.40

To verify H3, we reviewed the subjects' briefings after each reflection stage. The details of the descriptions increased in the order of Reflection 1, 2, and 3, as a whole. Subject #1 described an impasse about Java methods at Reflection 1, while he described an impasse about methodization at Reflection 3. Subject #2 described the lack of understanding arrays at Reflection 1, while he described his concrete mistakes on processing Java arrays at Reflection 3. The other subjects also displayed similar behavior, describing details at Reflection 3, such as concrete mistakes and times in impasses, that were not found in the descriptions at Reflection 1. We consider that these evaluation results support H3.

We must consider that a small number of subjects may influence the accuracy of these verifications. Our discussions do not have sufficient reliability because we could not procure a sufficient number of subjects for the experiment. However, we believe continuous practice will suppress this matter. Based on these discussions, we conclude that although preliminarily, our system could support learners to perform reflective learning as a whole.

6. Conclusion

In this paper, we described automatic, rule-based impasse detection and the system supporting learners to perform reflective learning by visualizing the summary of the detected impasses. Learners' impasse logs, which included the type of the impasse, the snapshot of program-code and the compilation and run-time errors at that time, would be useful for the learning and teaching of programming. Unlike learners coding effortlessly, those reaching an impasse are expected to experience some indication of the impasse during their coding process. Therefore, we developed a ruleset to detect impasses based on the indications in learners' codes and errors, and improved the detection ability by adjusting rules and threshold parameters through an experimental introduction into actual classrooms. Using this framework, we developed the system supporting learners to reflect how they had reached the impasse and how they had resolved it using the detected impasses as a clue. We conducted a small pilot experiment to evaluate the contribution of our system to reflective learning. The evaluation was based on a questionnaire survey and the achievements of reflective learning externalized by the subjects. The positive results of the evaluation suggest that our system could help learners attain some progress in reflective learning.

We must consider that a statistically insufficient number of the subjects in the experiment may influence the accuracy of verification. However, we believe that a continual practice of using our system will suppress this matter. Currently, our actual classrooms do not provide learners the time to reflect on their learning. We plan on introducing our system into the classrooms to make students cultivate a better understanding of programming exercises by performing reflective learning using our system. Persistent evaluations of students' understandings would clarify the contribution of our system, and hence suppress this matter.

Currently, the impasse summary visualizer is implemented only for learners' reflective learning. In the future, we will continue the development of the visualizer for teachers' one-on-one instruction during the exercise and for classroom designing. We believe that teachers will not need only an overview of the impasse detections, but an overview of the correspondences between the learning target and each detected impasse. We plan to extend our system to address this issue, incorporating existing program evaluation systems. The existing systems (Konishi, Suzuki, & Itoh, 2000) evaluate the correspondences between learners' code and sample code prepared by the teacher. Using this framework, we consider that our system could derive the correspondences between the learning target and detected impasse by integrating the learning target into each fragment of the sample code.

Acknowledgements

This study was supported by JSPS KAKENHI Grant Numbers JP24300282, JP16K01084.

References

Biswas, G., & Sulcer, B. (2010). Visual exploratory data analysis methods to characterize student progress in intelligent learning environments. *2010 International Conference on Technology for Education (T4E)* (pp. 114-121). IEEE.

Brown, N. C., & Altadmri, A. (2014). Investigating novice programming mistakes: Educator beliefs vs. student data. *Proceedings of the tenth annual conference on International computing education research* (pp. 43-50). ACM.

Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). What would other programmers do? Suggesting solutions to error messages. *Proceedings of the SIGCGI Conference on Human Factors in Computing Systems* (pp. 1019-1028). ACM.

Konishi, T., Suzuki, H., & Itoh, Y. (2000). A method of automated evaluation of learners' programs for assisting teachers. *IEICE Transaction on Information and Systems*, 682-692. IEICE.

Mazza, R., & Dimitrova, V. (2004). Visualizing student tracking data to support instructors in web-based distance education. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters* (pp. 154-161). ACM.

Piech, C., Sahami, M., Koller, D., Cooper, S., & Bilkstein, P. (2012). Modeling how students learn to program. *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 153-160). ACM.

Thomas, R., Kennedy, G. R., Draper, S., Mancy, R., Crease, M., Evans, H., & Gray, P. (2003). Generic usage monitoring of programming students. *Proceedings of the 20th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education* (pp. 715-719). ASCILITE.