Kashihara, A. et al. (Eds.) (2024). Proceedings of the 32nd International Conference on Computers in Education. Asia-Pacific Society for Computers in Education

Code Tracing Support Environment Based on Visualization of Cooperative Behavior of Multiple-Flows

Yasuhiro NOGUCHI[a*], Kotaro SUNAMA[a], Satoru KOGURE[a], Raiya YAMAMOTO[b], Koichi YAMASHITA[b] & Tatsuhiro KONISHI[a]
[a]*Faculty of Informatics, Shizuoka University, Japan*
[b]*Faculty of Business Administration, Tokoha University, Japan*
*noguchi@inf.shizuoka.ac.jp

Abstract: Several studies have been conducted to improve program comprehension through code tracing. However, most of these studies have focused on a single flow of control, with insufficient proposals to support program comprehension of multiple flows of control. It is difficult for novices to trace code with multiple flows in the correct order. In this study, we propose a code trace support environment for letting learners to trace the appropriate flow in execution with synchronization constraints. It also supports the component-based comprehension of synchronization mechanisms in multiple flows. The evaluation experiment confirmed the learning effect of understanding the appropriate order of steps. However, several issues regarding the component-based comprehension have been identified.

Keywords: programming, learning support system, visualization, multiple programs, abstraction, granularity

1. Introduction

Code tracing is essential to program comprehension in programming education. However, novice learners frequently struggle with tracing and program states (Sorva et al., 2013) (Lister et al., 2004). One cause of these difficulties is novice learners' low abstraction ability. Vainio and Sajaniemi (2007) indicated that the inability to increase the level of abstraction for data structures and/or operations in codes causes a depletion of working memory, resulting in mental simulation errors. Therefore, numerous research projects have supported learners in tracing program codes using program visualization techniques. Muldner et al. (2022) reviewed program visualization tools and their effectiveness.

Some modern software applications solve problems not only by a single sequential flow of control but also by executing multiple flows simultaneously. For instance, in computer science curricula, learners learn multiple flows in socket programming, parallel programming, and other programming and algorithm classes. Although it is more difficult for learners to correctly trace multiple flows, most learning support environments only allow tracing programs with a single sequential flow of control. Several studies have suggested visualization tools for learners to observe program concurrency such as Trümper et al. (2010) and Malnati et al. (2008), Lönnberg et al. (2011) and Strömbäck (2022). These tools are useful for programmers who have certain programming/debugging skills and knowledge for concurrency and synchronization, and they have already understood the behaviors and required function of each flow. However, these are not sufficient scaffolds for novices to learn a single flow of control to understand code with multiple flows.

In this study, we propose a learning environment that supports learners in tracing program codes with multiple flows. The proposed environment supports learners in correctly tracing multiple control flows and observing interactions among flows. We experimentally evaluated the learning effects of the proposed environment using an echo-back server and client applications, which are often assigned to network programming classes.

2. Learning Environment

2.1 Learning Methodology

In program tracing, learners trace program code step-by-step and observe the changes in program states between steps. For a single sequential flow, it is clear for learners that the flow of control must be traced; for multiple flows without dependencies, learners can also observe the result of each flow individually. Learners can also use existing visualization tools that support a single side-by-side control flow. In contrast, the multiple flows with dependencies, learners must consider the order of the steps themselves. If a subtask requires data to proceed, the flow must wait until the other tasks prepare the data. For instance, in program code that handles input/output devices, including network sockets, i/o blocking features synchronize multiple flows.

Such synchronization features are essential for programs with multiple flows; however, it is difficult for learners to understand them. First, in tracing the codes, learners should decide which flow they should move forward in every step. In this time, for choosing correct step, learners require to understand the state of the current flow (e.g., blocked or ready) and which flow will solve the blocked state in the later step. Second, these types of features are sometimes implemented as system libraries. Such codes are generally too complicated for learners, although they must understand the basic functions and behaviors of APIs required to trace the flow of the program. However, it is not necessary to trace the steps inside the libraries with the same granularity when tracing the application code and observing the details of the data at each step. Figure 1 shows the types of step granularities for a tracing function-call hierarchy. Most learning support systems enable learners to trace program codes based on trace type (A) or (B). However, it is complicated for learners to trace the code inside system libraries of trace types (A). If the function of the APIs can be explained only by the change between the before and after states of the function call, such as a pure function, (B) style tracing can be used for code comprehension. However, understanding the behavior inside libraries is important because these functions are designed based on side effects. For instance, the behavior inside “send” function make effect for the behaviors of other application; the behavior inside “listen” function make effect for the behavior of “accept” function as other function in network libraries. Thus, learners should proceed with style (C) tracing by choosing the essential steps/data in the library codes and interpreting complicated data structures at an appropriate level of abstraction.

As previously discussed, learners attempting to trace programs with multiple flows require the following support functions.

A) Guiding learners in tracing code with multiple flows by following the appropriate order of steps under the constraint of synchronizing features.
B) Guiding learners in tracing only the essential steps of the code inside APIs.
C) Visualizing abstracted behavior only between essential steps inside APIs.

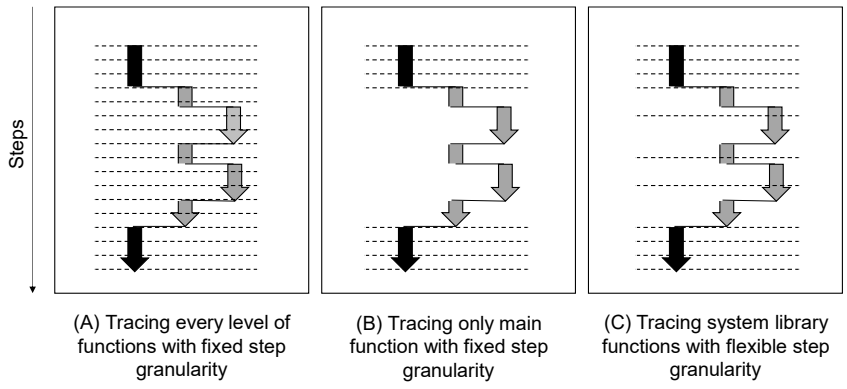

Figure 1. Step granularity for tracing the function-call hierarchy in a flow.

2.2 Architecture

2.2.1 Basic Architecture of TEDViT

In this study, we solved Problems A), B), and C) by extending TEDViT. TEDViT (Yamashita et al., 2013) is a program visualization environment that supports multiple panes: source code, domain world, and other dynamic information based on code execution. When learners trace program codes, they can observe program behaviors in the domain world synchronized with their chosen steps. Figure 2 shows the basic architecture of the TEDViT environment. When teachers provide program codes for an exercise and visualization policy as rules using an authoring tool, TEDViT generates the learning environment for novice programmers based on the program execution logs and the rules, and current steps of the codes chosen by learners.

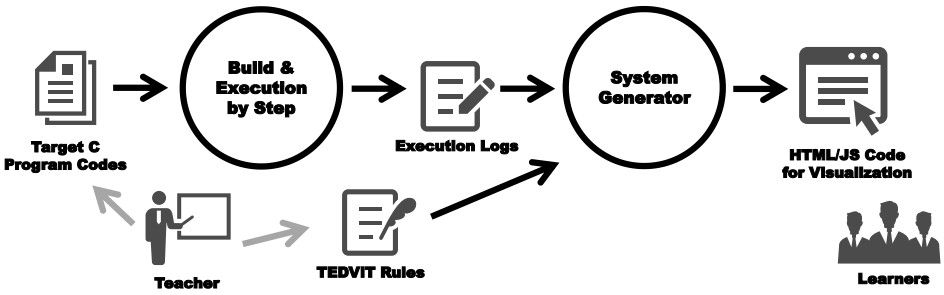

Figure 2. Overview of the architecture of TEDViT

2.2.2 Interface of the Extended TEDViT

Figure 3 shows the extended TEDViT interface, which includes two code views for each flow on both sides and a shared conceptual view at the center of the window. The conceptual view is shared by both flows; teachers can design all objects in the view by defining rules based on the intent of their instruction. There are four buttons at the bottom of each program code view that allow learners to step-forward/backward through the program code. In addition, an indicator notifies the learners of whether they can step forward.

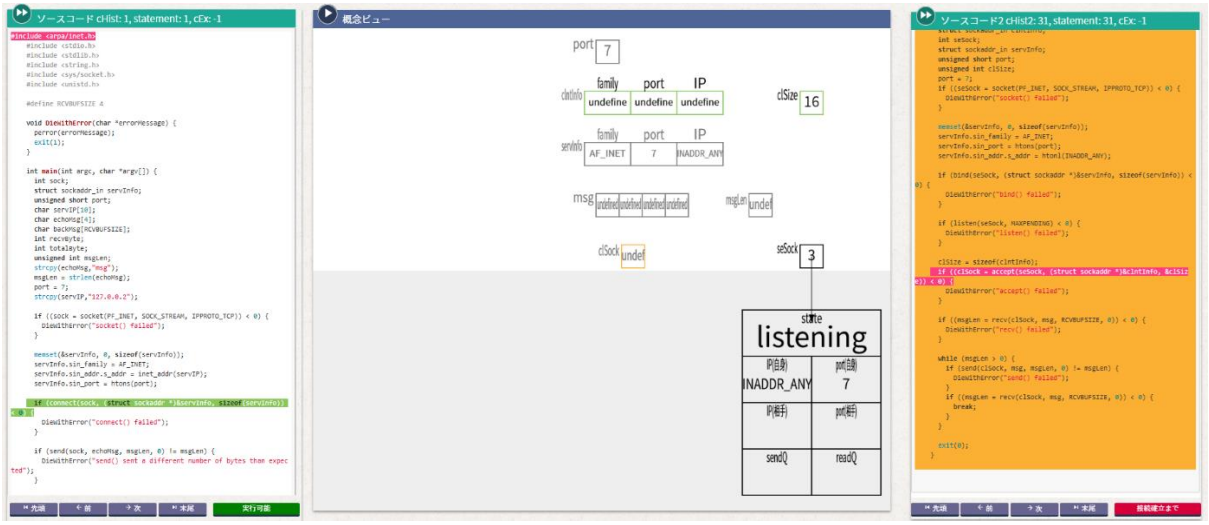

Figure 3. Interface of extended TEDViT

In this case, the left view shows the code for the client application, while the right view shows the code for the server application. The learner progressed through the server code to the statement of the “accept” function (pink line) by observing the changes in the conceptual view. At the bottom right of the conceptual view, a server-side socket data structure was visualized using TEDViT rules. Although the socket structure and other variables defined in

the actual code are complicated, the visualized socket data structure was abstracted based on the teachers' intention to focus on learners' understanding.

At the "accept" step, the "accept" function allows the application to block conditions; the environment disables the learner's step-forward action on the right code view and notifies the learner that the blocking condition continues until the client's connection is invoked at the step on the client code (green-colored line). The learner manipulates the step on the client code from the beginning (pink-colored line in the left view) to the step of the "connect" function (green-colored line in the left view). While the learner steps forward on a few inner steps of the "connect" function, the learner can observe the change in the "state," "IP," and "port" in both sockets in the conceptual view and the released condition in the server code view. In addition, if the learner steps forward on the inner steps of the "accept" function in the server code, they can observe the creation of a new socket for the connected client, and the local variable "clSock" links to the socket. In this manner, even for codes with multiple flows, the extended TEDViT allows learners to learn by comparing each step of the code and the change in conceptual view when tracing the code.

2.2.3 Visualization for Blocking/non-blocking Conditions

In this study, we treat echo-back server and client applications as cases of code with multiple flows. Figure 4 shows the possible execution times of the blocking and releasing functions. To merge each execution log with the correct step order, we extended the format of the execution logs to record the actual step time before each step. Based on the actual time both before and after the step, the extended TEDViT reconstructed the appropriate order of steps for learners, such as [D] to [X] or [F] to [X].

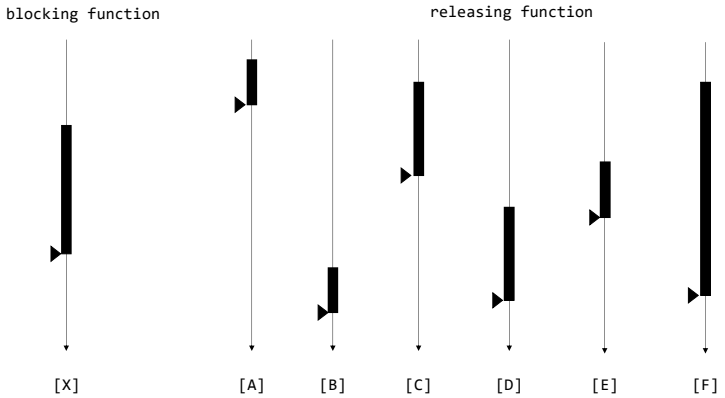

Figure 4. Cases of execution time between the blocking and releasing functions.[1]

By allowing learners to trace multiple flows in considering their blocking/non-blocking conditions, the extended TEDViT allows teachers to define blocking and releasing functions using rules. Teachers can define combinations of blocking and releasing functions, as shown in Figure 5, by setting the line number for each code and/or regular expression to identify the function call. The extended TEDViT supports learners in tracing the appropriate order of steps based on these definitions and reconstructed execution logs. This definition states that when the learner reaches a step of the block function, it prohibits the learner's step-forward action in the flow and encourages them to step forward in the flow on the release function side, based on execution logs. Moreover, when the learner reaches the release step, the corresponding blocking function is not reached in the other flow, prompting the learner to step forward in the flow on the block function side. By following these guidelines, learners can trace multiple flows of code based on the actual executed flow; they can also trace multiple flows of code without deviating from the constraints imposed by the block-release function relationship.

[1] The case [B] is the situation that releasing function in [B] did not work for releasing the blocking condition in [X], because the releasing function in [B] was called after the end of the blocking function in [X]. Thus, for learners, the appropriate order of the steps is from [X] to [B].

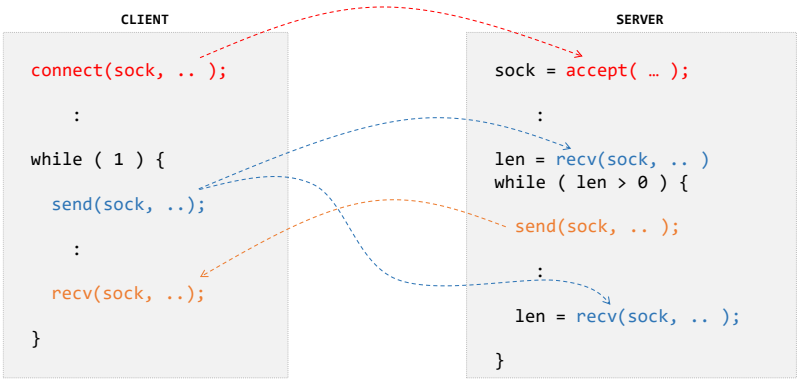

Figure 5. Defined relationships of blocking and releasing functions

3. Evaluation

3.1 Procedure

In this evaluation, the following two hypotheses were proposed.

H1: Extended TEDViT supports learners' understanding of multiple flows in programs better than exercises with ordinary IDE and textbooks.
H2: Extended TEDViT supports learners' understanding of libraries' functions and behaviors better than exercises using ordinary IDE and textbooks.

We recruited 11 undergraduate and 3 graduate students majoring in computer science. We confirmed their basic programming skills and applied our pre-test to balance the control and experimental groups. This experiment was designed with 20-minutes pre-test, 25-minutes exercise, and 20-minutes post-test. During the exercise, the experimental group used the proposed system, and the control group used a basic IDE. Both groups referred to the same textbook, which contained explanations of the server/client programs and libraries. In the pre-test and post-test, we asked 19 questions. We categorized eight questions into C1, regarding the understanding of the flow of the sample codes, and the other 11 questions into C2, regarding the understanding of the function and behavior of the socket APIs.

3.2 Result & Discussion

Table 1 lists the differences between the pre- and post-test assessments of the learning effect scores of the groups. Regarding H1, it was confirmed that learners could understand multiple flows in the programs in the proposed environment. All subjects in the experimental group had higher scores on C1 questions. This indicates that the subjects succeeded in tracing multiple flows of applications with the appropriate order of steps. Regarding H2, it was not confirmed whether learners understood the functions of library APIs and their inner behaviors. The medians of both groups were the same, but the mean of the control group was higher than that of the experimental group. This indicates that the proposed environment may be the cause of the learners' incorrect understanding.

Table 1. *Results of learning effect in the experiment*

Category	Group	N	Mean	Median	S.D.	S.E.
C1 questions	Control	7	1.14	1.00	1.46	0.553
	Experimental	7	2.86	2.86	1.21	0.459
C2 questions	Control	7	3.00	3.00	1.29	0.488
	Experimental	7	1.86	3.00	2.12	0.800

Table 2. *Results of two-tailed Mann-Whitney U Test for learning effects*

Category	U	p	Mean Difference	Effect Size
C1 questions	8.50	0.043	-2.000	0.653
C2 questions	17.50	0.386	1.000	0.286

4. Conclusion

In this study, we proposed a code-tracing support environment for programs with multiple flows, and confirmed the effectiveness of the environment in understanding multiple flows in programs. In the experiment, the proposed environment worked effectively to comprehend the behavior of multiple flows of control and synchronization. In addition, learners acquired an understanding of the system APIs required to trace the behavior of multiple control flows. However, for some learners, the proposed environment was not sufficient to deepen their understanding of the inner behavior of system APIs. In the future, for program comprehension in tracing code with functional components, it will be necessary to improve the supporting function to visualize the behavior of the component with appropriate abstraction.

Acknowledgements

This work was supported by JSPS KAKENHI (grant number 22K12311).

References

Lister, R., Adams, S, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J, E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. In Working group reports from ITiCSE on Innovation and technology in computer science education (ITiCSE-WGR '04), 119–150.

Lönnberg, J., Malmi, L., & Ben-Ari, M. (2011). Evaluating a visualisation of the execution of a concurrent program. In Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11), 39–48. https://doi.org/10.1145/2094131.2094139

Malnati, G., Cuva, C. M. & Barberis, C. (2008). JThreadSpy: A tool for improving the effectiveness of concurrent system teaching and learning, International Conference on Computer Science and Software Engineering, 549-552, doi: 10.1109/CSSE.2008.11.

Muldner, K., Jennings, J., & Chiarelli, V. (2022). A review of worked examples in programming activities. ACM Transactions on Computing Education, 23, 1, Article 13 (March 2023), 35 pages. https://doi.org/10.1145/3560266

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. ACM Transactions on Computing Education (TOCE), 13(4), 1–64

Strömbäck, F., Mannila, L., & Kamkar, M. (2022). Pilot study of progvis: A visualization tool for object graphs and concurrency via shared memory. In Proceedings of the 24th Australasian Computing Education Conference (ACE '22), 123–132. https://doi.org/10.1145/3511861.3511885

Trümper, J., Bohnet, J., & Döllner, J. (2010). Understanding complex multithreaded software systems by using trace visualization. In Proceedings of the 5th international symposium on Software visualization (SOFTVIS '10), 133–142. https://doi.org/10.1145/1879211.1879232

Vainio, V. & Sajaniemi, J. (2007). Factors in novice programmers' poor tracing skills. In Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '07), 236–240. https://doi.org/10.1145/1268784.1268853

Yamashita, K., Suzuki, M., Kito, Y., Suzuki, Y., Kogure, S., Noguchi, Y., Yamamoto, R., Konishi, T. & Itoh, Y. (2023). Interaction support systems between teachers and visual content for effortless creation of program visualization, Research and Practice in Technology Enhanced Learning (RPTEL), 18(33), 1–31.