Three Phase Self-Reviewing System
for Algorithm and Programming Learners

Tatsuhiro Konishi, Hiroyuki Suzuki, Tomohiro Haraikawa and Yukihiro Itoh
Faculty of Informatics, Shizuoka University, Japan

Abstract: This paper introduces an electronic report submission system that helps effective learning of algorithms and programming. It proposes a three-phase reviewing system that involves self-reviewing of algorithms, self-reviewing of programs and staff reviewing. This is an improvement of our existing two-phase reviewing system that only supports the latter two phases. In the additional phase for algorithmic checking, learners describe an algorithm graphically using PAD, compile it, and execute it to verify their algorithm first without being troubled by syntax of a programming language; this supplies effectiveness to the efficient self-reviewing system.

Keywords: Algorithm, programming, learning, self-review system.

1. INTRODUCTION

This paper introduces an electronic report submission system that helps the learning of algorithms and programming. Learners' programs typically contain numerous mistakes and must be reviewed again and again before becoming acceptable. While it takes a few hours until the learners can get staff comment, the turnaround time must be shortened, so as not to distract the learner's concentration. Thus we have proposed two-phase reviewing: an automated self-reviewing phase for improving efficiency of learning, and a careful reviewing phase by staff.

Although algorithms should be represented independently of any specific programming language, present algorithm education is filled with language-dependent explanations and practices. In such a situation it is doubtful that learners can be conscious of the algorithm itself and some researchers claim that teaching of algorithms and of programming should be separated (Crews 1998). A flowchart or a PAD (Program Analysis Diagram) is used for representing algorithms and there are tools for editing and executing algorithms (Maezawa 1984, Hitachi Systems & Services), however, most of them still depend on a specific programming language. Therefore we

develop a language-free algorithm representing system and an algorithm validation support system, and propose a method of algorithm education using these systems.

We constructed a three-phase reviewing system that involves self-reviewing of algorithms, self-reviewing of programs, and staff reviewing, by improving our existing two-phase reviewing system that only supports the latter two phases. Through our practical operation, we learned that learners tend to be hooked on syntax when teachers let them check their code. We are now implementing a new system that also allows an algorithmic check before coding. Learners can graphically represent, compile, and execute their language-free algorithms to verify them without being troubled by syntax of a programming language; this supplies effectiveness to the efficient self-reviewing system.

In this paper, we first introduce our two-phase reviewing system. We then discuss extension of the system, especially methods of assisting algorithm education and algorithmic check.

2. TWO-PHASE REVIEWING SYSTEM

We had already adopted a self-reviewing system in programming education to reduce the turnaround time. At first, we thought that a GCC compiler was sufficient for learners to review their programs locally if its warning level is maximized. However inspection of submitted reports reveals that many typical mistakes are not detected as errors or do not receive warnings. For example, an erroneous code "**if (1 <= month <= 12) { … }**" never receives a warning since it is considered as a condition that compares logical value of **1 <= month** (0 or 1) and an integer value **12**. The condition perfectly satisfies C syntax. Such kinds of mistakes are left uncorrected until staff notice and write a reviewing comment. The fact leads to a long turnaround time and heavy staff loads. Fortunately we know some code reliability checkers for embedded systems. For continuous fault-free operation required for embedded systems, the tools perform strict source-level analysis to point out any doubtful scraps. Some of them find meaningless conditions or operators (such as **return r++;** for a local variable **r**), make a string comparison using an operator ==, and even find typical array-index overruns. We applied such tools to the collected reports, and employed one product for self-program-reviewing in our two-phase system.

2.1 SELF REVIEWING PHASE

As we supposed, the reliability checker was useful but it was sometimes unusable for educational use due to too many suggestions or too few detected mistakes. Also, it has a user interface for professional use. So we decided to make a '*wrapper*' of our reliability checker which can both provide flexible levels of suggestions and a user-friendly interface.

We designed an email-based report submission system which sends back compilation status and source-code reliability analysis immediately. The email report should consist of the report text as an email body and sources (and headers) as attachment files. When a **recv** script receives a report via a Mail Transfer Agent (MTA), it tears the attachments off and forks to the gcc and the reliability checker in this order. Subsequently, it sends back the result by email. The wrapper is designed to suppress or replace some over warning suggestions for untrained programmers. The wrapper suppresses all strong suggestions about Y2K problems and code-optimizing directions, and replaces some suggestions with [*Info*]s. An original [*Info*] for "Line 11" in Fig.1 was "*Using a pointer for accessing to array "**month[i]**" instead could generate smaller or faster object code.*", which is from the viewpoint of the embedded tools.

A learner who receives a modified suggestion as a reply can re-submit their report, and repeat this self-reviewing process depending on their need. A learner can also browse their submission history, every automatic reply and additional reviewing comments from a staff including scores (described in the next subsection) at a web site.

[Reviewer's Comment] (handtyped)
You must verify the behavior of your program before submitting your report.
[Compilation Errors and Warnings]
None. (Congratulations!!)
[Suggestions by Reliability Checker]
Line10: Wrong || usage between "**1** <= **i**" and "**i** <= **12**."
Line9: **for** statement contains wrong comparison "**i** > **0**."
Line11:"**month[i]**" overruns because 12 is assigned to "i" by a **for** statement at line9.
Line11: [Info] You may use a pointer for accessing to array "**month[i]**."
Line14: Using an increment or decrement operator to the return value "**i**++" makes no effect.
Line7: Variable "**month**" is not referenced.

Fig. 1: an Example of Feedback (translated)

2.2 STAFF REVIEWING PHASE

Since reports are usually refined repeatedly via a self-reviewing process, the staff only have to review their best reports; this greatly reduces the reviewers' task. The reviewing screen is separated into two panes: the first one displays scoring buttons, a comment field and a contents selector. The second one initially displays a summary composed of the compilation status, reliability analysis and body part of a report. It also shows any source file selected by a staff. Comments and scores are immediately reflected on the web site and learners can submit their reports again at this point, too.

3. THREE-PHASE REVIEWING SYSTEM

We are constructing an improved system based on a three-phase model. The self-reviewing phase of the previous model is now divided into *self-program-reviewing* and an additional phase named *self-algorithm-reviewing*. The ideal flow is given below.

The first phase: The phase helps learners to fix their algorithm. A web-based *algorithm editor* enables learners to represent algorithms independently of any specific programming language. A submitted algorithm is compiled on a server by an *algorithm compiler*. Learners can download the object code to execute and verify the algorithm. Learners can repeat this phase to make their algorithms accurate.

The second phase: This phase helps learners to verify their programs by themselves. A learner writes his/her program in C language at this phase, and then submits it with an algorithm representation created in the first phase. If the program is successfully compiled, a *correspondence checker* verifies whether the submitted program is implemented correctly in accordance with the associated algorithm. In addition to the reliability status, learners can also inspect the correspondence via a web-based *correspondence viewer*.

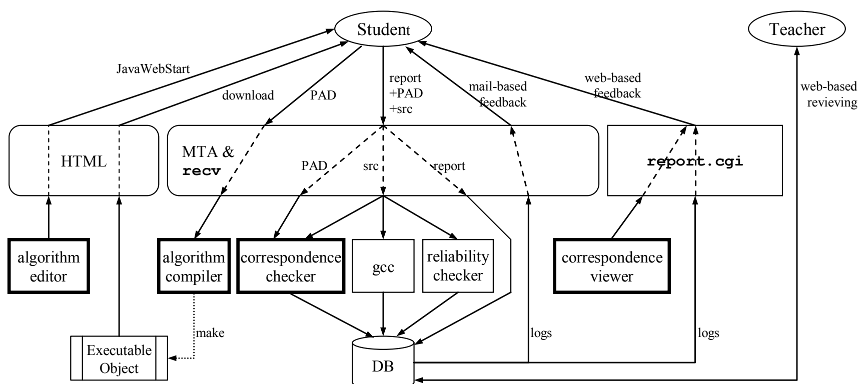

Fig. 2: Three-phase reviewing system

The third phase: This phase provides staff hand typed comments to learners. It is similar to the staff reviewing of the two-phase system, except that the phase now has a potential to provide additional information to the staff; correspondence between submitted source code and a standard algorithm written by staff.

We designed this three-phase system (Fig.2) that employs the four components mentioned above: an *algorithm editor*, an *algorithm compiler*, a *correspondence checker*, and a *correspondence viewer*, in addition to the previous two-phase system. We have constructed these components as distinct four assisting systems: a language-free algorithm representing system, a language-free algorithm validation support system, an algorithmic checker, and the two-phase reviewing system. We adopt the language-free

algorithm representing system as the *algorithm editor* (Shinmura 2003). The main feature of the language-free algorithm validation support system is used for the *algorithm compiler*. The algorithm checker works as the *correspondence checker*.

3.1 SELF ALGORITHM REVIEWING

(1) Algorithm editor and Algorithm representation

Our *algorithm editor* adopts PAD representation. It has functions to help users to edit PAD expression easily. Additionally, it should be able to provide an appropriate operation set to users. In order to decide the set, we discuss representation policy for each operation. The representation of an operation has to satisfy the following requirements.

i. Learners can write algorithm by using the representation without learning any specific programming languages.

ii. The representation includes no ambiguity.

iii. The granularity of the operation should be controllable. Too large granularity of an operation allows a learner to jump into the goal using too few operations. On the other hand, if the granularity is too small, a learner can't represent his/her algorithm intuitively.

iv. It has levels of both concrete and abstract representations. One of the essential aims of algorithm education is to make learners learn how to grasp problem solving procedures in an abstract level. For example, linked lists can be represented by structures and pointers in C language. In concrete level, operations on the linked list are described by such terms as "pointer", "structure" and so on. However, learners should consider the solving process abstractly by using terms like "link", "node". So, both concrete and abstract words should be provided to describe algorithms.

Solution for i and ii: When someone describes an algorithm by any formal languages, they have to study notations and the grammar of the language. In order to avoid such extra work, we adopt natural language as a method of describing an operation. However, unrestricted natural language may be ambiguous, so we restrict the vocabulary and the sentence pattern. For learner's convenience, we prepare acceptable sentences as templates, and let learners select a template from a menu.

Solution for iii: Appropriate granularity of description depends on the goals of exercises. Therefore our system allows staff to select an appropriate granularity by selecting available templates for each exercise.

Solution for iv: In order to let learners represent algorithms abstractly, the *algorithm editor* provides templates which correspond to abstract operations to abstract data structures. We surveyed explanations of algorithm in textbooks of programming and found 7 typical data structures used to describe algorithms abstractly; list, binary tree, table, stack, heap, matrix and queue (Suzuki 2001). Based on the survey, we prepare templates to represent algorithm abstractly. When a staff member intends to let

learners represent their algorithms abstractly, they select such templates as mentioned above.

Fig.3 shows a screenshot of our *algorithm editor*. In Fig.3, (1) is the area for algorithm editing. An example of an algorithm representation is displayed. (2) shows the list of variables, and (3) is the reduced drawing of (1). In the area (1), learners draw algorithm representation by mouse operation, menu selection and keyboard input.

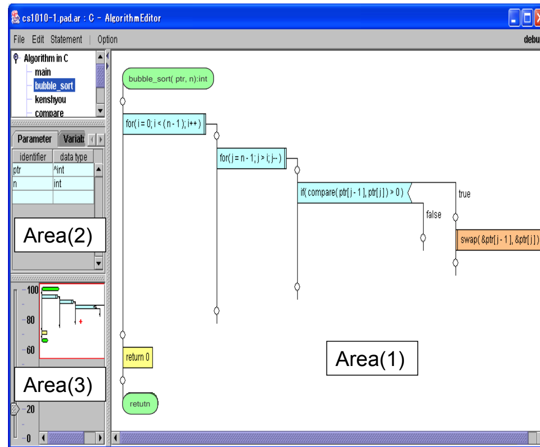

Fig.3: Screenshot of the algorithm editor

(2) Validating algorithms by learners

In order to make it possible to validate algorithms by learners, we have to develop the function of executing represented algorithms (*algorithm compiler*). In our previous work, we constructed a system which converts abstract representations of operations into source codes in a specific programming language (Suzuki 2001). We use the system as the *algorithm compiler*. With the *algorithm editor* and the *algorithm compiler*, a learner can review their algorithms in the following way: First, a learner downloads the *algorithm editor* from the web. Next, he/she writes his/her algorithm by the editor and saves it as an algorithm file. If he/she wants to execute the algorithm, he/she submits the algorithm file to our server. Then our system compiles the algorithm, and creates an executable file. The learner can locally execute the algorithm and can also validate its behavior.

3.2 SELF PROGRAM REVIEWING

In the second phase, a learner implements the validated algorithm using a programming language, in order to acquire knowledge on syntax of the programming language and techniques on implementation.

We think that mistakes in an erroneous program can be categorized into two types. One is caused by misunderstood syntax or mistyping. The other is caused by the fact that a learner can't break down an operation in algorithm into smaller pieces, or can't convert operations into a set of statements of a programming language. Compilers and code reliability checkers can only check the former mistakes. The latter can be checked by comparing a

learner's algorithm representation with his/her source code. The method of checking such correspondence is as follows (Suzuki 2001):

The *correspondence checker* breaks down an operation into the smallest grain-sized operations, which are comparable with statements of a programming language. When there are some operations represented abstractly, many possible candidates can be generated from them. The checker searches for the candidate most similar to the learner's program. Through the searching, the checker stores information of correspondence between operations in the algorithm representation and statements in the learner's program.

By using these components, a learner reviews their programs as follows: at first, a learner writes a program based on their validated algorithm and submits both the algorithm file and C program to our server. Then gcc, the code reliability checker, and the *correspondence checker* work. Diagnoses by the components are stored in a database and are immediately sent to them by email. Additionally, they can see the correspondence between their algorithm and program by using the *correspondence viewer* that works on a web browser (Fig.4). Learners can easily find operations/statements which do not correspond to the program/algorithm. In addition, when a learner places a mouse cursor on an operation/statement, the statement/operation which corresponds to it changes its color. By these functions, learners can confirm whether they correctly implement their algorithms.

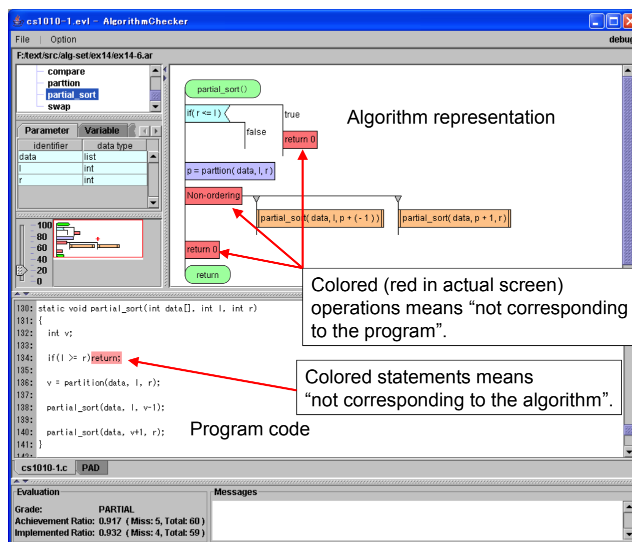

Fig.4: Screenshot of the correspondence viewer

3.3 STAFF REVIEWING

Finally, a staff member reviews programs, algorithms, and reports which are submitted by email. The staff can refer to all the diagnoses given to the learners. Moreover, they can use the *correspondence checker* and the *correspondence viewer*, in order to compare a learner's program with a

standard algorithm that they write. Such usage makes it easier to find bugs which are not found by learners.

4. CONCLUSIONS

We proposed a self-reviewing system that realizes efficient and effective learning of algorithm and programming. The self-reviewing system is a front-end of our three-phase electric report reviewing system. The new first phase, *self-algorithm-reviewing*, allows learners to concentrate on representing their language-free algorithms in a PAD before writing their programs. An algorithms sent to the server is internally translated into C language and compiled. The system makes the object code downloadable by learners. Learners can repeatedly submit, validate and correct their algorithms by themselves. Learners write their codes in the second phase of *self-program-reviewing*. Formally, the phase includes not only a syntax and reliability check, but also self-correspondence-check between a learner's algorithm and his/her program. The third phase newly provides a staff with detailed analysis report; that will be of great help in scoring or writing hand typed comments. Now we are planning to apply the system to actual classes of algorithm and programming in our university.

5. REFERENCES

T. R. Crews, U. Ziegler (1998): The Flowchart Interpreter for Introductory Programming Courses. Proceedings of FIE '98 Conference, pp.307-312.

H. Maezawa, M. Kobayashi, K. Saito, Y. Futamura (1984): Interactive system for structured program production, Proceedings of the 7th international conference on Software engineering, pp.162-171. Florida, United States.

Hitachi Systems & Services, Ltd.: TOPITAL: PAD CASE tool, http://www.hitachi-system.co.jp/topital/index.html

Hiroyuki Suzuki, Takaomi Sakai, Tatsuhiro Konishi, Yukihiro Itoh (2001): Automated Evaluation of Learner's Pro-grams by using Algorithm Representations Independent of Programming Languages, Proceedings of ICCE2001, vol2, pp.883-890.

K.Shinmura, E.Iida, H.Suzuki, T.Konishi, and Y.Itoh (2003): The Method to Support Algorithm Learning without Being Distracted by Programming Lan-guages, Proceedings of ICCE2003, pp.1210-1214.