Designing an Automatic Debugging Assistant for Improving the Learning of Computer Programming[1]

Maria S. W. Lam[*], Eric Y. K. Chan[#], Victor C. S. Lee[*], and Y. T. Yu[*]

Department of Computer Science, City University of Hong Kong
[*]{marialam, csvlee, csytyu}@cityu.edu.hk, [#]chanyk@cs.cityu.edu.hk

Abstract. Finding bugs in programs (debugging) is a core skill for practical programmers. However, debugging programs can be difficult to novice programmers. Even worse, repetitive failures may defeat students' enthusiasm for learning. The presence of a mentor giving hints and help face-to-face with students will surely make such a learning process much more effective and enjoyable. However, this requires lots of manpower and resources. To address this problem, we seek to capitalize on the potential advantages offered by hybrid learning. We are working towards a system for providing a certain level of automatic debugging assistance to students. Instructors can identify common errors in students' programs using the system and incorporate useful debug-guiding information into it so that students will be prompted with pertinent hints when common errors are detected in their programs.

Keywords: automatic debugging assistant, computer programming, PASS, test cases and annotations.

1 Introduction

Finding bugs in programs (debugging) is a core skill for practical programmers. However, debugging programs can be difficult to novice programmers. Even worse, repetitive failures may defeat students' enthusiasm for learning. The presence of a mentor giving hints and help face-to-face with students will make such a learning process much more effective and enjoyable. However, this requires lots of manpower and resources. To address this problem, we seek to capitalize on the potential advantages offered by hybrid learning (also called blended learning) [1], whereby students' learning experience through their face-to-face interaction with instructors and tutors is supplemented and enriched by the use of e-learning systems.

We are working towards an Automatic Debugging Assistant (ADA) which aims at providing a certain level of automatic debugging assistance to students. Instructors can identify common errors in students' programs using ADA and incorporate useful debug-guiding information into it so that students will be prompted with pertinent hints by ADA when common errors are detected in their programs.

[1] This work is partially supported by a Teaching Development Grant (project no. 6000145) from City University of Hong Kong. Corresponding author: Victor C. S. Lee.

ADA is designed to be an extension of an existing Programming Assignment aSsessment System (PASS) developed at our university for improving the teaching and learning of computer programming [2]. Since 2004, PASS has been used in many computer programming courses to automate the programming assignment submission and grading process [3-5]. Through PASS, the instructor may upload the assignment and practice problems with some preset public test cases for students to obtain the problem specification and test their programs online. Students can submit their programs to PASS for assessment before the submission deadline specified by the instructor. Afterwards, upon the instructor's request, PASS can automatically produce the results of assessment of the students' programs [5]. Then students can read their grades, together with the feedback manually added by the tutor.

The sections that follow in this paper will explain the advantages and challenges of providing automatic debugging assistance for students. Then we discuss some relevant attributes of a test case, which is the core data entity of the automatic debugging assistant, and describe the design framework of the system. These are followed by a case study, some observations arising from it, and finally, some concluding remarks.

2 Problems in Manual Debugging

Solving problems by constructing correct computer programs is an iterative process. It is common to take more than a few iterations to debug and test programs before a correct version can be produced. However, debugging programs can be difficult, time-consuming, and prone to human errors. More bugs may be inadvertently introduced while revising the program if the original bugs cannot be correctly located and fixed during debugging.

There are many integrated development environments (IDEs) which provide facilities to computer programmers for software development. An IDE normally consists of a source code editor, a compiler and/or interpreter, build automation tools, and (usually) a debugger. They are designed to maximize programmer productivity by providing a user-friendly integrated environment with different components for users to edit, compile, build (create the executable program) and debug their programs. For debugging, the IDEs mainly help the programmers in removing the typing or syntax errors in the source code. However, the most difficult step in debugging is to find the logical software bugs which prevent the computer program from behaving as intended. Conventionally, the capability for detecting and fixing logical bugs depends heavily on the student's own experience, logical thinking ability and programming skills.

Practice is a major learning activity to gain experience and develop good programming skills. So, it is very important to arouse students' interest in doing more practice in a computer programming course. However, students nowadays will easily lose motivation and interest in program debugging, especially when they do not know if they are working in a right direction. An automatic debugging assistant, whose design is presented in the next section, will probably help motivate students to maintain the momentum of the search for a solution to their programming tasks.

3 Design of the Automatic Debugging Assistant

3.1 Attributes of a Test Case

A set of carefully designed test cases not only helps students partially verify the correctness of their programs, but also helps them identify possible semantic errors by comparing the program outputs with the expected ones. In addition, in case of wrong program outputs, meaningful annotation describing each test run can help students figure out the source of errors more efficiently. In a testing and debugging process, test case is a core data entity. We first discuss some relevant attributes of a test case.

Level of Difficulty. PASS allows the instructor to prescribe programming exercises and inform the students of the level of difficulty of each exercise. To extend this concept, the difficulty of developing a solution to the same programming exercise can vary substantially by prescribing different test cases that the student's program is required to pass. Thus, instructors can design and classify the test cases into different levels of difficulty so that students can construct their programs progressively and incrementally, and test their programs at each stage of development [4]. Moreover, students can easily learn that they are working in a right direction and get a sense of satisfaction when their programs can at least pass the test cases at a lower level of difficulty. This motivates them to put more effort to get their programs pass all the test cases at the higher levels of difficulty.

Visibility. Instructors can specify whether a test case is *open/public* or *hidden/private* [5]. For instance, in PASS, test cases for practice problems can be open for students (that is, all students know exactly what these test cases are) to check the correctness of their programs, while for assessment problems, like assignments or online quizzes, instructors can hide all or some of the test cases for the purpose of grading [2]. In another programming submission and testing system called Marmoset, Spacco et al. [6] further divide test cases into four visibility levels to serve different purposes, namely, *student tests* (those written by students), *public tests* (provided to students), *release tests* (selectively made available to students) and *secret tests* (not disclosed to students until after the submission deadline).

Inputs. Inputs are values to be fed into a student's program during execution. Depending on the problem specification, instructors may ask students to write programs which accept a single input, a specified number of inputs or an unlimited number of inputs until a specific value is entered or a specific key is hit. Also, whether any pre-processed input validation is required depends on the learning objectives of the problem. It is crucial to design a wide diversity of test inputs to cover most, if not all, possibilities to test the target program comprehensively. On the other hand, redundant test inputs that hint at the same bug may be eliminated to

avoid a waste of debugging time. Nevertheless, it should be borne in mind that a program passing all the test cases can still be incorrect [2].

Expected Outputs. To check the correctness of students' programs, each test case is associated with a corresponding expected output. In this respect, PASS can be configured to display messages with varying amount of details. One option is to simply tell whether student's program output matches the expected output or not. Another option is to compare student's program output with the expected one and highlight their differences to provide more hints for students to figure out the possible source of errors [4].

Common Wrong Outputs with Annotations. Although there can be many different wrong outputs produced by different incorrect programs when they are executed with the same test case, some wrong outputs caused by typical program bugs usually appear more frequently than those caused by less common bugs. Our aim is to provide help to students who commit the common mistakes, as such help will have greater chances of benefiting more students.

In a recent study, Jadud [7] explored the behaviour of first-year university students who were learning to program in Java, and came up with a list of the most common errors they encountered related to the compilation of their programs. The list includes *unknown variable*, *missing or misplacement of semicolon*, *incorrect matching of brackets*, *unknown method*, and others. In another study, Morimoto et al. [8] developed a system, called TeCProg (which is a support system for Teaching Computer Programming), that analyzes the trend of past compilation errors to facilitate teachers to understand the common mistakes made by students in their programming work. In our work, we are more interested in detecting the logical errors made by students. To identify these errors, we examined a sample of incorrect programs submitted by students in their first programming course. Based on these sample programs, we prepared some annotations as debugging hints specifically for each of the bugs. Table 1 shows some typical program bugs that result in common wrong outputs and the corresponding debug-guiding annotations. Given the bug types, specific annotation for each test case can be entered manually by instructors or retrieved automatically from a repository. In the latter case, instructors can, if desired, further refine the retrieved annotations to provide more concrete hints to students to help them debug their programs.

General Annotation for Uncommon Wrong Outputs. It is impractical to capture all possible wrong outputs produced by incorrect programs when they are executed with a particular test case, as these programs can be written in myriads of ways. Besides the common wrong outputs, other wrong outputs or unexpected behaviour can be produced by an erroneous program, such as the lack of output, timeout, deadlock, or memory/resource leakage. These errors can only be identified by examining the program code line by line. In this case, the annotation may simply document the purpose of the test case as a hint for students to dig into the cause of bugs by themselves.

Table 1. Some typical program bugs

Type	Description	Annotation
Declaration	A declaration specifies the interpretation and attributes of a set of identifiers, which are used to define data types and initial values for variables, functions or constants in the source code. Many errors are caused by incorrect declaration statements. Example. Using an integer-type variable to store a real number may induce mathematical errors. Say, when the number 1.99 is stored as an integer, its value may be mistaken as 1.	Verify that the variables, functions and constants are correctly defined and initialized with appropriate data types and values.
Operator / Specifier Symbol	Many errors occur when the operators / specifier symbols are wrongly used.	Verify the use of operators / specifier symbols.
Boundary	Many errors occur when the input values are at the border of different parts of the input domain.	Verify or add codes to handle inputs at the border of different parts of the input domain.
Conditional Statement (if-then-else)	A conditional statement performs different computations or actions depending on whether its condition evaluates to true or false.	Verify the conditions in conditional statements.
Iteration	Errors commonly occur in loop constructs. In particular, failing to test the loop termination condition correctly may lead to an infinite loop or an incorrect number of iterations.	Verify the loop termination conditions.
Arithmetic Rounding	Rounding error is the difference between the calculated approximation of a number and its exact mathematical value. Example. To calculate (1/3 + 1/3) to 2 decimal places, if the value 1/3 is rounded to 0.33 before addition, the result will be 0.66. Otherwise, if rounding is done after addition, the result will be 0.67.	Verify the mathematical equations or calculations to avoid errors due to rounding.
Output Format	Incorrect formatting of the output, such as incorrect number of decimal places, wrong spelling of words, or occurrence of a redundant punctuation mark.	Verify that the format and wording of the program output conform exactly to the requirements stated in the program specification.
Exceptional / Abnormal	Some errors may occur when the program fails to handle gracefully the abnormal values, such as zero or negative values.	Verify or add exception value handling code, if applicable.

3.2 Design Framework of ADA

ADA is designed for use in the following way. Students submit their programs to PASS for testing. If a submission passes all the open test cases, PASS will prompt a successful execution message. Otherwise, ADA will provide the test case annotations together with a failure message. These annotations are either manually pre-set or derived from past submissions. Meanwhile, the current submissions, together with their wrong program outputs, will be recorded and subsequently become instances of past submissions for future annotation enhancement. In addition, submission statistics such as the number of attempts before successful execution will be recorded for analyzing the performance of students, the common mistakes committed by students, the level of difficulty of the problem or its associated test cases, and so on.

The design framework of ADA is shown in Fig. 1. It consists of six modules: *Test Case Editor / Upload*, *Result Processor*, *Submission Statistics Collector*, *Submission History Accumulator*, *Submission Post-processor* and *Annotation Repository*. The modules are briefly described as follows.

Test Case Editor / Upload Module. This module provides an interface for instructors to edit or upload test cases in a predefined format.

Result Processor Module. This module compares student's program outputs against expected outputs. For each correct program output, a successful execution message will be displayed. Otherwise, a failure message and an annotation for each wrong output will be provided. Depending on the preference of instructors, the expected outputs can be shown with the differences highlighted to provide more hints for students to revise their programs.

Submission Statistics Collector Module. Students are allowed to attempt and submit programs for testing any number of times before a given deadline. This module collects submission statistics such as the number of attempts before successful execution and the mean time between submissions. It also provides functions for instructors to organize and analyze the statistics for system performance evaluation.

Submission History Accumulator Module. All information in each submission, including the test cases, student programs and their outputs, will be recorded by this module for subsequent processing to identify typical bugs and derive respective annotations.

Submission Post-processor Module. This module analyzes the historical submission records and extracts useful knowledge or rules by artificial intelligence or data mining techniques to identify typical program bugs and derive respective specific annotations. It is a recursive process. The number of submission records will grow with time and

more information can be used to refine the rules and enhance the quality of annotations or even track the trend of typical program bugs.

Annotation Repository Module. This module stores the typical program bugs and the associated specific annotations derived by the Submission Post-processor Module. This information can be accessed by the Result Processor Module to retrieve a particular specific annotation given a bug type.

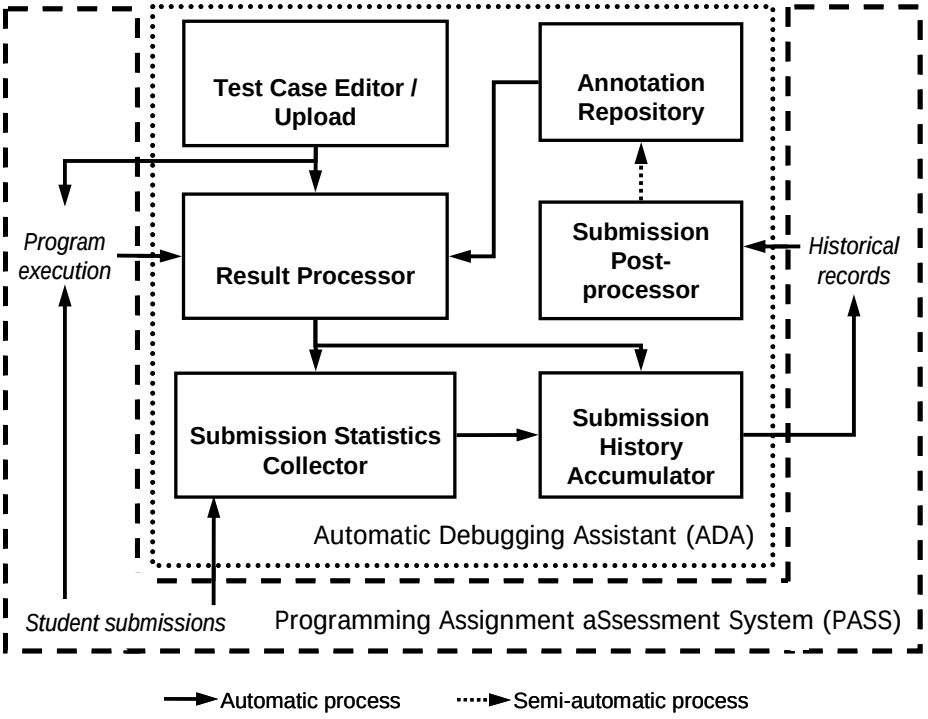


Fig. 1. Design Framework for Automatic Debugging Assistant

4 Case Study

To demonstrate the effectiveness of the output-specific annotation about helping students to find out the bugs of an incorrect program, a case study has been undertaken. In the rest of this section, the specification of the selected problems in a programming course are given first, followed by the creation of test cases and their annotations for the problems, and finally some observations and discussions.

4.1 Selected Programming Problems

We selected a foundation C programming course for the case study. Most students in this course are new to programming. In the course, students need to learn the basics of C programming, including data declaration, use of operators, as well as conditional statements and loops. During the learning process, they often feel frustrated to tackle the semantic errors in their programs, the issue which we are trying to address.

From the course's historical records in PASS, we selected the following three problems for detailed study, based on their complexity as well as their expected outputs' characteristics. Problems with too simple output(s), say, output values that are "Yes or No" or "True or False", are not selected. Problems with historically the highest attempt rate and failure rate from the remaining ones are selected. Thus, we selected firstly, the problems that contain various patterns of outputs, and secondly, the problems which expose the difficulties encountered by most students in their learning.

Problem 1: Use of output formatter

Write a C program, **`bmi.c`**, to get the *weight* (in kilograms) and *height* (in meters) of a person. Then calculate and print the *Body Mass Index* (*BMI*) in 2 decimal places according to the formula: *BMI = weight* / (*height* * *height*).

A sample output of the program follows, showing in what way the program is expected to interact with the user. In presenting these sample outputs, we shall adopt the convention that all input values to the program are underlined.

```
Enter your height in m: 1.8
Enter your weight in kg: 60
Your BMI is 18.52
```

Problem 2: Use of mathematical operators

Write a C program, **`convertTime.c`**, that reads the number of seconds and converts it to hours, minutes and seconds. A sample output of the program follows.

```
Please enter the number of seconds: 5000
5000 second(s) =  1 hour(s) 23 minute(s) 20 second(s)
```

Problem 3: Use of loop

Write a C program, **`factor.c`**, that reads a positive integer n and outputs all its factors k, where $1 < k \leq n - 1$. A sample output of the program follows.

```
12
2 3 4 6
```

Below is another sample output of the program. Note that since the integer 7 has no factors other than 1 and itself, the program should produce no output.

```
7
```

4.2 Test Case and Annotation Creation

Because a variety of programming styles may be used by different students, it would be difficult to design test cases based on their program source code. Therefore, test cases for each problem are designed by considering the input and output domains. Since input validation is not required in these problems, invalid inputs are omitted. Test cases and their corresponding annotations are created based on *Table 1*.

One class of typical test cases consists of exceptional cases to determine whether the programs can handle unusual values. For instance, if the input value of *weight* in Problem 1 is assigned to zero, the output *BMI* should be 0.00 (except when zero is assigned to the variable *height*). This test case can verify the correct substitution of the numerator and denominator. Stack overflow will occur if the program wrongly swapped the two variables. Similarly, if a program for Problem 3 did not treat zero as a special case before executing the while loop, the program may not terminate.

Another class of typical test cases consists of boundary test cases, where the input value(s) is (are) at the border of different parts of the input domain, such as an input of 60 or 3600 seconds, or inputs at the transition of seconds to minute or minutes to hour conversion in Problem 2. If the mathematical operators, such as division '/' and modulo '%' operators, have been wrongly used, the resulting number of seconds or minutes can become equal to or greater than 60, which is anomalous.

Normal test cases are used to determine whether the programs perform the normal computations correctly. Although a variety of outputs can result when the logic of the program is wrong, some specific errors will trigger certain definite wrong outputs. For instance, some students wrongly used an integer variable to store the input value of *weight* and a non-integer variable to store the input value of *height*, or vice versa.

There are two types of annotation which may provide assistance to students: *output-specific annotations* and *general annotations*. The creation of output-specific annotations is based on analyzing the wrong output corresponding to the test case. If the wrong output can be classified as a common wrong output, an appropriate message will serve as debug-guiding annotation to hint where the source code may get wrong. For instance, "check the correct position of the numerator and denominator" message is a specific annotation for the "stack overflow" exception output in Problem 1. For outputs that cannot be classified as common wrong outputs, stating the objective of the given test case will assist students to debug their programs. The general annotations are independent of the actual output produced by the student's program. For instance, one example of general annotation for Problem 3 can be "This input does not have any factor". The student may follow this reasoning to check why his/her code produces any output at all.

4.3 Observations

Fig. 2 to 5 show, respectively, some sample output messages for the chosen problems. The columns of each message include the serial number of the test case, the inputs, annotation, actual output and result. For simplicity, the expected output is omitted.

Fig. 2 shows the result of an erroneous program when 1.01 is input as the value for *height* and 62.99 as the value for *weight*, respectively, in Problem 1. Here, the program fails because the data type of the variable *weight* was wrongly declared as `int` (an integer variable), instead of `float` (a floating point variable) as stated in the annotation. Students can then check the code in the data declaration parts to debug their programs.

Fig. 3 shows an error case for Problem 2, in which the program fails to convert 60 minutes to 1 hour. Corresponding to such an output, ADA will produce a specific annotation "1 minute instead of 60 seconds and 1 hour instead of 60 minutes" to remind students to check the conversion formula.

Fig. 4 shows a "Time limit expired" error for Problem 3 caused by infinite loop when the input value is zero. The annotation "Check input 0 causes infinite loop" suggests the student to review the looping condition. With the hint of annotation, the student should be able to consider zero as one of the cases to terminate the while loop in his/her program.

Fig. 5 shows another programming error for Problem 3. The problem requires the program to output all factors of an integer except 1 and the integer itself. However, the input value itself is also printed in the output list. The specific annotation "No need to check equal to input in while loop condition" hints that the termination condition of the while loop is wrong.

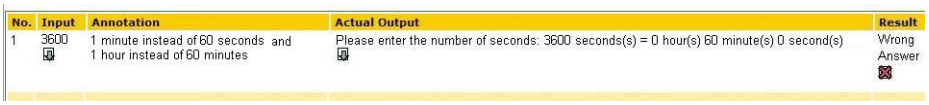
4.4 Discussion

It is difficult to trace a semantic error in computer programs even for an experienced programmer. There is no debugger which is intelligent enough to tell the programmer why the program output is wrong. It would be hard to guess how to trace such an error as the program code listing grows longer and longer. ADA utilizes historical data to generate useful annotations for different kinds of common logical errors. It definitely is not intended to be fully automated. However, with the help of annotations as debugging hints, students can locate the bugs more effectively if such bugs have been committed by students in the past.

Our initial experience in the case study suggests that, with the assistance of test case annotations to serve as debug-guiding information, students can more precisely identify the location of bugs in their programs so that they do not need to rewrite the whole program again and again. Conversely, without the assistance of annotations, students may as well attempt to rewrite the programs and subsequently other new bugs may be introduced, making the problem even harder to solve. Worse still, students may eventually be fed up with frustration and abort their attempts.

No.	Input	Annotation	Actual Output	Result
1	1.01 62.99	The numerator should be declared as float instead of int	Enter your height in m: Enter your weight in kg: Your BMI is 60.78	Wrong Answer

Fig. 2. ADA prompts declaration error with annotation (Problem 1)



No.	Input	Annotation	Actual Output	Result
1	3600	1 minute instead of 60 seconds and 1 hour instead of 60 minutes	Please enter the number of seconds: 3600 seconds(s) = 0 hour(s) 60 minute(s) 0 second(s)	Wrong Answer

Fig. 3. ADA prompts boundary value error with annotation (Problem 2)

No.	Input	Annotation	Actual Output	Result
1	0	Check input 0 causes infinite loop		Time limit expired

Fig. 4. ADA prompts exceptional case error with annotation (Problem 3)

No.	Input	Annotation	Actual Output	Result
1	12	No need to check equal to Input in while loop condition	2 3 4 6 12	Wrong Answer

Fig. 5. ADA prompts loop condition error with annotation (Problem 3)

5 Conclusion

This paper has described the design of ADA, an automatic debugging assistant that is an extension of PASS, a Web-based automatic programming assignment assessment system. Students can use PASS to test and verify their programming assignments online. The newly extended debugging assistant, ADA, aims to further relieve the workload of tutors in guiding students to find bugs in their program code. ADA is not intended to be intelligent enough to understand programs. Rather, it provides a platform for tutors to consolidate the causes of common programming errors and transform such information into helpful hints. This kind of automatic debugging assistance is expected to supplement our daily face-to-face teaching, hence realizing the potential benefits of hybrid learning in computer programming courses.

Enhanced by ADA, PASS will enable the instructor to return instant debug-guiding information to students. Initial responses from the users in our pilot case study are encouraging, and further work is underway to perform a systematic evaluation of the effectiveness of such a feedback mechanism and the debug-guiding information.

References

1. Graham, C.R., Allen, S., Ure, D.: Benefits and Challenges of Blended Learning Environments. In: Khosrow-Pour, M. (ed.) Encyclopedia of Information Science and Technology, pp. 253--259. Hershey, PA: Idea Group (2005)
2. Yu, Y.T., Poon, C.K., Choy, M.: Experiences with PASS: Developing and Using a Programming Assignment aSsessment System. In: 6th International Conference on Quality Software (QSIC 2006), pp. 360—365. IEEE Computer Society Press (2006)
3. Chong, S.L., Choy, M.: Towards a Progressive Learning Environment for Programming Courses. In: Cheung, R., Lau, R., Li, Q. (eds.) New Horizon in Web-based Learning: Proceedings of the 3rd International Conference on Web-based Learning (ICWL 2004), pp. 200--205, World Scientific Publishing Co. Pte. Ltd (2004)
4. Choy, M., Lam, S., Poon, C.K., Wang, F.L., Yu, Y.T., Yuen, L.: Towards Blended Learning of Computer Programming Supported by an Automated System. In: Workshop on Blended Learning 2007, pp. 9--18, Prentice Hall (2007).
5. Choy, M., Nazir, U., Poon, C.K., Yu, Y.T.: Experiences in Using an Automated System for Improving Students' Learning of Computer Programming. In: 4th International Conference on Web-based Learning (ICWL 2005), LNCS, vol. 3583, pp. 267--272, Springer (2005)
6. Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J.K., Padua-Perez, N.: Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In: 11th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'06), pp. 13--17, ACM Press (2006)
7. Jadud, M.C.: Methods and Tools for Exploring Novice Compilation Behaviour. In: 2nd International Computing Education Research Workshop (ICER'06), pp. 73--84, ACM Press (2006)
8. Morimoto, Y., Kurasawa, K., Yokoyama, S., Ueno, M., Miyadera Y.: A Support System for Teaching Computer Programming based on the Analysis of Compilation Errors. In: 6th International Conference on Advanced Learning Technologies (ICALT'06), pages 103--105, IEEE Computer Society Press (2006)