What They Really Do? Attempting (once again) to model novice programmers' behavior

Vassilios Dagdilelis
Dept. of Educational and Social Policy University of Macedonia
156 Egnatia Str., Thessaloniki,Greece
+30-310-891336
dagdil@uom.gr

Maya Satratzemi
Dept. of Applied Informatics University of Macedonia
156 Egnatia Str., Thessaloniki,Greece
+30-310-891897
maya@uom.gr

Georgios Evangelidis
Dept. of Applied Informatics University of Macedonia
156 Egnatia Str., Thessaloniki,Greece
+30-310-891844
gevan@uom.gr

Categories and Subject Descriptors: K.3.2 [Computers and Education]: Computer science education.

General Terms: Algorithms.

Keywords: student strategies; development and validation of programs; compiler generated error messages.

In the last two decades, a large amount of research has been conducted in an effort to form a model of student behavior when they try to solve algorithmic or programming problems. The construction of the model is based on the analysis of many types of data, such as for example: (a) the characteristics of the programming languages the students work with, (b) the strategies of the solution that the students follow, and (c) the characteristics of the proposed problem. However, we must observe that modeling is often not based on long-term observations of actual teaching and the pro- posed problems are usually quite simple.

In this paper we attempt to examine a variety of aspects of students' behavior when they learn to program. More specifically, we study: the strategies students use in order to develop and validate a pro- gram; the possible role of students' errors in the development of their programs; and the methods students use to deal with these errors. The study was carried out on 90 second-semester CS stu- dents who worked in pairs during the 2-hour lab session. They were given a brief description of the Binary Search algorithm and were asked to implement it using AnimPascal. In this study we present the results we obtained from the analysis of the successive versions of students' programs. Based on these results we propose teaching methods to help students overcome the difficulties they face when they learn programming.

The majority of the programs were developed using two methods: a) mental execution of the algorithm; and b) initial development of a skeleton program corresponding to the algorithm and extension of it to a complete program through trial and error attempts (elimina- tion of syntax errors by using code correction or by bypassing the erroneous code and the corresponding erroneous results). Empirical data enable a clear distinction to be made between students into *quickers* and *slowers*: in the first category are placed those students who react quickly to the erroneous results of their attempts and locally correct the error (adhesive plaster method) seeking simply to attain a correct result, whereas, in the second category a time period devoted to investigation elapses between message and reaction. In many cases the initial formulation includes the preliminary sections of the program. In other cases the initial skeleton of the code is not complete and in fact is not possible to "pass" through the compiler. The students' strategy in this situation is to correct the program locally in order to retest it. The difference between this and the previous case is due to the fact that students do not appear to follow some general plan but simply proceed step by step.

The strategies of identifying a prototype structure and its corresponding processing or that of tailoring were observed in very few cases. We assume that the identification of prototypes requires a deep knowledge and understanding of the prototypes, which our students did not possess. For the same reason, they did not use tailoring methods, since their use requires that the programmer knows how to solve sub-problems. Our results indicate that we should systematically train our students so they become capable of identifying prototypes and of using standardized problem solving methods.

Regardless of the methodology applied, the development and vali- dation of the student programs was affected in a great degree by the syntactic errors they made and the comprehension of the compiler generated error messages. The understanding of the messages ap- pears to be related to the degree of mastering the students have on the language they use. The understanding of the messages also depends on the actual message itself since system-generated mes- sages are often typically correct but hard for a novice programmer to understand. In the majority of the cases, messages for fatal errors result in the local correction of the code, in an attempt to eliminate the error message. The quickers try to make the error message dis- appear, whereas the slowers attempt to make sense of the meaning of the error messages and to react accordingly. The low degree of mastering of the programming language used, often forces the stu- dents to devise programming tricks in order to bypass the problem created by a syntax error. In many cases, the students were led to logical errors because of their inability to correctly interpret the compiler messages and correct the syntactic errors they had made. The most significant errors, however, arise from the loop testing conditions. As the successive versions of student programs reveal, the final determination of the correct Boolean expression was achieved only after a series of trial and error attempts of all the possible cases.

Considering the above results, we plan to direct our attempt to the following two axes: a) improve the messages generated by the AnimPascal compiler; b) propose exercises to students that contain syntax errors they will have to correct. This type of exercises will include control and repetition structures, nested if statements and compound Boolean expressions. We believe that these exercises will familiarize students with the corresponding error messages. We make such a proposal because during our programming course we gave emphasis to problems that could help students acquire prob- lem-solving skills rather than problems that could systematically familiarize them with erroneous code.