Evaluation and Assessment of Effects on Exploring Mutation Testing in Programming Courses

Rafael A. P. Oliveira
Dept. of Computer Systems
University of São Paulo
São Carlos, Brazil
rpaes@icmc.usp.br

Lucas B. R. Oliveira
Dept. of Computer Systems
University of São Paulo
São Carlos, Brazil
IRISA – Univ. of South Brittany
Vannes, France
buenolro@icmc.usp.br

Bruno B. P. Cafeo
Opus Research Group
Software Engineering Lab
Informatics Department – PUC-Rio
Rio de Janeiro, RJ, Brazil
bcafeo@inf.puc-rio.br

Vinicius H. S. Durelli
Dept. of Computer Systems
University of São Paulo
São Carlos, Brazil
durelli@icmc.usp.br

***Abstract*—Mutation analysis is a testing strategy that consists of using supporting tools to seed artificial faults in the original code of a software under test, generating faulty programs ("mutants") that are supposed to produce incorrect outputs. Novice programmers suffer of a wide range of deficits due to defective training processes. We argue that the incorporation of experiences on mutation testing in programming courses adds valuable knowledge to the learning process. In this paper we evaluate the effects of using mutation testing to improve the learning process of students in programming courses. We present results of experiments and analysis involving undergraduate students. These experiments are the continuation of a previous work in which we raise empirical evidences that the adequate incorporation of mutation testing in programming courses contributes to form an effective environment that fosters learning. To do so, we provide a mutation testing tool to promote the practice of mutation testing by novice programmers. Through practical experiences and several analysis survey we measured the effects of using the mutation testing criterion to teach programming. In addition, we collected the opinion of senior students who already knew mutation testing concepts about their opinion on the usage of mutation concepts to teach novice programmers. Our findings reveal that the effective use of mutation analysis concepts contributes to the learning process, making students see the code as a product under development that is the result of a careful manual coding process which they need for measuring and predicting the effect of each command. The main contributions discussed in this paper are: (1) presenting results of an empirical analysis involving undergraduate students, thus giving us preliminary evidence on the effects of the novel practice; (2) exposing possible practices to explore mutation testing in programming classes, highlighting the limitations and strengths of such strategy; and (3) a mutation testing tool for educational purposes.**

***Keywords*-mutation testing; experimental study; computer-aided instruction.**

I. Introduction

Currently, in the information technology area, there is a high demand for highly skilled developers. There are several vacancies in the technology field due to the lack of well-prepared professionals. Part of this problem can be associated to "poorly" designed programming courses. Often, traditional programming teaching methods may not meet all students' expectations [1]. Novice programmers suffer a wide range of deficits and professional limitations due to defective training processes [2]. Then, novice students have not had enough exposure to programming languages to develop strategies for solving problems in a computer programming setting [3]. Even competent programmers make certain mistakes that can be used as experience to improve the learning processes. Among these mistakes, one is the off-by-one error, which happens, for example, when an iterative loop that was meant to iterate *'n'* times iterates either *'n – 1'* or *'n + 1'* times. These errors arise when programmers use the wrong relational operator or fail to note that a sequence starts at zero rather than one.

A possible way to improve the learning process of novice programmers is by the practice of introducing software testing activities in the classroom [4]. Teaching testing techniques and criteria in the very beginning of programming courses may represent an effective practice to improve students' skills [5]. Conceptually, Software Testing is the dynamic process of executing a software product (or a software module) aiming at checking whether it corresponds to its specifications, considering the environment in which it was designed [6]. Testers must use testing techniques and criteria to achieve more effective testing. Testing techniques and criteria provide systematic ways to find conditions to raise the probability of finding errors. Several researchers have been studying the application of testing practices in conjunction with programming fundamentals in programming courses [5, 7, 4]. Implementing this practice may lead to gains under two aspects: (1) supporting students to learn technical programming language concepts more quickly, and (2) creating the habit of applying testing techniques in their systems under development earlier.

Despite the fact of several researchers have been dedicated efforts on raise evidences on using testing concepts to support teaching programming, there are few studies on the effects of combining specific testing criteria and programming courses. In this context, in a previous study [8], we have presented a prototype of a mutation testing tool specially designed to supporting the learning process. *Pascal mutants* supports the mutation testing criterion of Pascal programs, being extremely useful to novice programmers. Mutation Testing [9] is a fault-based testing criterion that consists of introducing small syntactic changes into a *Software Under Test* (SUT), creating slightly different versions (so-called mutants) in order to obtain a test suite capable of identifying all of the seeded faults (small

978-1-4799-8454-1/15/$31.00 ©2015 IEEE

errors) [9, 8]. Then, the tester has to analyze defective codes, providing test cases to produce distinct outputs for the SUT and its mutants [9]. Mutants are often similar to the faulty programs written by novice students in programming courses. However, the cost of mutation testing has traditionally been so high it cannot be applied without full automated tool support.

The main goal of this paper is to provide preliminary evidences and discussions on the effects of exploring the criterion mutation testing to teach programming foundations/-fundamentals to novice programmers. To do so, we have conducted an experiment on using *Pascal mutants* and mutation testing in an undergraduate-first-year class. We have observed the students behaviors when in contact with the mutation testing criterion. Then, we have measured the effects on using mutation testing through a detailed questionnaire to assess the student's level of understanding over specific fragments of code. In addition to that, aiming to assess the opinion of senior undergraduate students enrolled in a software testing course, we have prepared a proper survey to assess their opinion about the premature contact with mutation testing by novice programmers. Empirical and survey-based studies have raised evidences the usage of mutation testing might improve the learning process under different aspects such as better general grasp levels and better understanding the abstract concepts that may result in the improvement of the professional skills. Then, the contributions associated to this paper are: (1) raise empirical evidences on the effects of exploring mutation testing criterion to teach programming; (2) highlighting the limitations and strengths of mutation criterion to teach novice programmers; and (3) a mutation testing tool for educational purposes. The main implication of our exploratory study is the indication of the importance of introducing software testing concepts to teach novice programmers.

II. Related Work

Several researchers have been exploring the idea of using software testing integrated to programming courses in order to improve the learning process. Most of these approaches are associated to functional testing, which mean, they are not related to source code analysis. However, we consider these studies related to our research because they look to educational purpose of the usage of software testing. The main difference between our study and the group of studies presented below is the fact that we are interested on measure the educational effects of a single specific testing criterion.

Jones [7] revealed evidences on the benefits of presenting testing concepts in conjunction to programming classes instead of separated courses. The stud sates three points: (1) software testing represents a good topic to be inserted in the regular curriculum of technology courses; (2) students need the practice of testing activities to improve their educational experiences; (3) there is a basic set of recommended testing skills; and (4) regular courses should impart testing experiences.

Brito et al. [1] present an approach in which test cases (test input and test output) are rearranged and reused to improve the quality of activities performed by novice programmers in programming courses. Empirical studies involving about 60 undergraduate students show that when students have some previous implemented test cases to evaluate their solutions, the quality of the programs produced by them increases significantly. Buffardi and Edwards [4] present an approach to quantitatively evaluate the effects on presenting software testing concepts to students. The study involves programs generated by 883 different students over five years. However, the study focuses on evaluating software testing behaviors that students exhibited in introductory computer science courses. Several (76%) of the students have designed different testing behaviors on different assignments, leading to good habits among students. Souza et al. [10, 5] believe that the integration of testing concepts and programming courses can improve the learning process in students. Then, the studies propose and validate a web-based tool that sets an environment in which students can submit their solution for a given problem, and then, their solution is "tested" against the reference program's.

III. Methodology

This section presents concepts and methodology for our investigation.

A. Mutation Testing

Mutation testing [9] is a fault-based testing criterion which relies on typical mistakes programmers make during software development. Mutation approaches seed defects into the SUT in order to examine its behavior. This criterion relies on the *Competent Programmer* and the *Coupling Effect* hypotheses. According to the former, the Competent Programmer's hypothesis states that due to the fact that programmers are competent, they write programs that are close to being correct. Therefore, it can be assumed that a program written by a competent programmer has simple faults; often, they are small syntactic changes. The Coupling Effect hypothesis states that complex faults result from the combination of small ones. As a result, fixing the small faults will probably solve the complex ones. Then, regarding the testing process, finding artificial defects allows the effective detection of real faults.

Mutation testing creates several versions of the SUT with slight faults. In practice, given an original program P, the criterion consists of creating a set of mutants, M, that are slightly modified versions of P. "Mutation Operators" define the modification that must be applied to P. Mutation operators are necessary to insert small defects into the original program, creating mutants. These defects are generated by using a set of simple syntactic rules. Then, for each mutant m, $(m \in M)$, the tester runs the test suite T originally designed for P. If there is a test case t, $(t \in T)$, and $m(t) \neq P(t)$, this mutant is considered dead. If not, the mutant is considered alive and the tester should improve T with a test case that reveals the difference between m and P. If m and P are equivalent, then $P(t) = m(t)$ for all test cases.

Creating mutants from an original program involves a set of rules established by mutation operators. These rules change the original program, creating mutants with syntactically valid

changes. Typical mutation operators modify expressions by replacing, modifying, inserting, or deleting either operators or variables. For instance, Figure 1 presents the original program and one of its mutants. In this particular case, a mutation operator replaces the logic operator (Line 3).

```
1 (**** Original *******)
2
3   if (a && b) then
4       c := 1
5   else
6       c := 0;
```

```
1 (**** Mutant *********)
2
3   if (a || b) then
4       c := 1
5   else
6       c := 0;
```

Fig. 1: Original and Mutant Source Code.

An undecidable issue in mutation testing comes out when the fault injection creates mutant programs that are functionally equivalent to the original program. Namely, original programs and *Equivalent Mutants* will always yield the same results. Detecting mutants that are equivalent to the original program is one of major obstacles for practical usage of mutation testing, requiring the full knowledge of tester who must perform this task manually through code analysis. However, using a mutation testing tool, once a tester has marked all possible equivalent mutants for an SUT, it is possible to compute the "mutation score". The mutation score is a coverage metric and it is expressed as the ratio of the number of dead mutants over the number of non-equivalent mutants. Thus, when the tester reaches a 100% mutation score, all non-equivalent mutants are said to be dead.

B. Pascal Mutants

Pascal μjutants implements the mutation testing criterion at the unit level of Pascal programs. *Pascal μjutants* is available online as an open source tool in a Sourceforge repository, along with its supporting documentation and several example programs. The link to access the tool is `http://sourceforge.net/projects/pascalmutants/`

1) General Structure and Wizard: In Oliveira et al. [8] we have defined a prototype of a mutation testing tool for educational purposes. In this study, we have significantly improved that prototype composing *Pascal μjutants* Through seven specially designed mutation operators, testers (students) can generate faulty versions of an original program code. Students can select specific mutant operators, create testing projects, manage and execute testing data, check the mutation score and compare original and mutant codes. Technically, the implementation of *Pascal μjutants* integrates six main modules: (1) a Pascal syntax analyzer and syntax tree generator, (2) a compiler and loader of mutants, (3) a test case manager, (4) a results evaluator (test oracle), (5) a mutant manager, and (6) a test reporter. Further technical details of *Pascal μjutants* can be found in our preliminary study [8] in which *Pascal μjutants* were presented as command-line based implementation.

Regarding the scope of this study, we have extended *Pascal μjutants* to work under a robust wizard that supports all of the functionalities presented previously. After results of some pilot studies, we have designed this wizard towards promoting the education purpose of mutation testing. Then, this wizard is important for novice programmers in order to explore mutation testing to understand parts of the source code. The wizard leads the programmer's focus to important parts of the code instead of mutation testing details. It has an intuitive and user-friendly interface. Through a set of UIs (*User Interfaces*) provided by *Pascal μjutants* students can perform several activities: (1) manage test cases, (2) analyze mutants and an original program, (3) mark equivalent mutants, and (4) visualize dead, alive, and equivalent mutants. Students can examine testing statistics (i.e., number of alive, dead, and equivalent mutants), including the mutation score. After defining a prototype for *Pascal μjutants* in Oliveira et al. [8], we have implemented *Pascal μjutants* in a way that it encourages the users to adopt an autonomous posture during the learning process.

2) Mutation Operators: Pascal μjutants implements seven mutant operators that are presented in Table I. These operators are not a complete set with regard to the identification of an ideal test suite for testing purposes. However these mutation operators are sufficiently complete to produce sets of mutants useful for educational purposes. In addition, the generic structure of the tool enables testers to add new mutation operators.

C. Mutation Testing Supporting Novice Programmers

Defining inputs that differ between SUT and mutants requires testers to fully understand the programs' data and information flow, from assignment of values to their use. Naturally, this full understanding depends on the specific program's control structure of the programming language in which the SUT is written. The process of reading and understanding source code may be trivial for advanced programmers, however it may be challenging for novice programmers. On the other hand, reading source code to perform mutation testing may represent a valuable process of learning for novice programmers. Section IV presents empirical evaluations on using mutation testing and *Pascal μjutants* in practical programming classes.

IV. Empirical Evaluation

Aiming at verify the practical usage of mutation and *Pascal μjutants* to support the learning process of novice programmers, we have conducted a two-fold independent evaluation:

(Analysis 1) A practical empirical evaluation of using *Pascal μjutants* in an undergraduate course on "Computer Programming" for first-year students. This experiment aims to provide a direct assessment on using mutation testing in a programming course for beginners (Subsection IV-A). We have compared the process of code analysis and understanding by students using mutation testing criterion and regular compilers; and

(Analysis 2) A survey analysis involving senior students in an undergraduate course of "Software Testing and Inspection". This survey aims at measuring the opinion of students who are familiar with mutation testing however had no contact with mutation concepts when they were novice programmers (Subsection IV-B).

TABLE I: *Pascal μutants* Mutation Operators.

	mutation operator	acronym	original code	mut. operator's effect
1	Change of Arithmetic Operator (for other arithmetic operator)	CAR	`c := a + b;`	`c := a - b;`
2	Change of Relational Operator	CRO	`if (a > b) then`	`if (a < b) then`
3	Change of Logical Operator	CLO	`if (a && b) then`	`if (a || b) then`
4	Adding "NOT" Operator in Control Expressions (if, while, etc)	ANOiCE	`if (a > b) then`	`if (NOT(a > b)) then`
5	Removing A Command	RAC	`a := b; b := c ;`	`a := b; ;`
6	Change for a Scalar Constant for its Successor and Predecessor	CSCSP	`const average = 5;`	`const average = 6;`
7	Change the "Repeat" Command for "While" and vice versa	CRCfW	`repeat a := a*count; count := count+1; until (count < 7);`	`while (count < 7) do begin a := a*count; count := count+1; end;`

We now provide a detailed view of each analysis: research questions, the methodology we followed, and the experimental/survey strategies we have designed.

A. Analysis 1 – a practical experience on using mutation by novice programmers

This study is designed to answer the following *Research Questions* (RQ):

RQ1: Can the mutation testing criterion facilitate the learning process of novice students in programming courses?

RQ2: What are the trade-offs and recommendations of using mutation testing to support the learning process in programming courses?

We have followed the same protocol established in a previous pilot study detailed in Oliveira et al. [8]. Then, we were able to correct some imperfections, such as the need for a background on software testing by the students. In general terms, the idea of the experiment is to provide the very first contact of novice students in a programming course with mutation testing. To do so, we use the resources and functionalities provided by *Pascal μutants*

The subjects students involved in the experiment consisted of 28 undergraduate students enrolled in the second semester course on "Computer Programming" UNESP campus Rio Claro, São Paulo, Brazil (*Universidade Estadual Paulista*). The set of students were enrolled in a Bachelor of Computer Science's course and they had a prior knowledge of about fifty-hours course of algorithms in Pascal language. Then, one can conclude that the students were considered novice programmers with no background on software testing neither software engineering concepts.

The experiment was conducted in framework of a four-hour course. We have started our experiment presenting theoretical concepts on Software Testing, including definitions and practical examples. Besides the traditional test phases (*Unit, Integration, and System*), the main testing techniques and their particular criteria were presented. Then, we have focused on the mutation testing criterion, detailing each foundation of the criterion. Finally, we have present *Pascal μutants* to the subject students. We presented them a step-by-step on how to create a testing project and how to load test data, identify equivalent mutants, and compare original and muted code. However, during this step of the experiment, the students had no practical contact with *Pascal μutants*

After the aforementioned experimental setup, we conducted a controlled experiment in laboratory. This experiment was composed by a dynamic activity exploring a simple program and a survey to assess the student's level of understating on the program. We then started evaluating the behavior of each student in order to contribute to answering the RQ 2. First of all, we have divided the subject students in two groups. Group 1 (G1) was designed to conduct the experiment using a regular Pascal Compiler. Group 2 (G2) was designed to conduct the experiment using *Pascal μutants* and exploring the concepts of mutation testing. Then, we have set G1 to be a *control group*, and G2 to be our *treatment group*. We gave no further explanations and advises to any group to avoid potential biases.

The activity assigned to the students consisted of analyzing a blind code, trying to understand its functionality through code analysis and running it against different inputs. We did not provide any specification about the program functionality. Equally, the identifiers assigned to each variable offered no clues about their usage. The code represented a program to figure the classical sequence of numbers named *Fibonacci series*. Starting from one and one, Fibonacci series is a sequence of numbers in which each subsequent number is the sum of the previous two, for example: "*1, 1, 2, 3, 5, 8, 13* ...". In our subject program, three parameters were accepted to represent, respectively, (1) the first element of the series, (2) the second element of the series, and (3) the limit of the series. During the previous classes, we did not offered prior contacts with the Fibonacci algorithm.

Aiming to provide empirical evidences to answer RQ 1, we elaborated a questionnaire that is able to measure the level of understating of the students. The questions were open about the program, possible specific changes in the source code, and they were designed to cover the students' attitudes toward: (1) general aspects about the program; (2) influence of the starting points in the algorithm; (3) influence of the stop conditions (ending points); and (4) relation among number of iterations and parameters. Table II presents all of the survey's questions.

Section V presents the results of this empirical evaluation.

B. Analysis 2 – a survey analysis with senior students

We believe, after having some experiences during an undergrad course, students are able to notice whether some activities could help them in some period of their student lifetime. Then, we designed a survey to assess the opinion

TABLE II: Analysis 1: Survey's Questions.

#	Questions
Q1	Describe what your thoughts are about the main functionality of this program?
Q2	Is there any difference if operator '$<=$' were used instead of operator '$<$' in the command '$while$'? Regarding the variables 'a', 'b', and 'c', in which situations may differences occur?
Q3	For the majority of inputs, would the number of interactions in the loop be different if the line 6 has the following command: '$b := 2 * b$'? Justify your answer using examples.
Q4	What would happen in the case where line 6 has the following statement: $b := b + a$;? Justify?
Q5	Once '$a = 3$', '$b = 5$' and '$c = 40$', how many terms would be printed in the program output if the operator '$+$' was replaced by '$*$' in the command '$while$' (line 1)? Justify?

of senior students about learning mutation testing during the very beginning of programming courses. Then, this analysis aims at identifying whether senior students, who are familiar with mutation testing, would advise novice programmers to practice mutation testing to improve their programming skills.

Unlike from the students involved in the previous experiment, the set of subject students that composed Analysis 2 consisted of 21 students enrolled in the undergraduate course on "Verification, Validation, and Software Testing" at USP (*University of Sao Paulo*). These students had prior knowledge of Software Engineering, concepts of Software Testing, and more than three programming language. Once the course is offered for a broad audience, the experiment involved undergraduate students from three different bachelor courses: (1) *Computer Science* (33%); (2) *Information Systems* (14%); and *Computer Engineering* (52%). Additionally, they had between 6 and 14 semesters of course, most of them having 8 semesters of course. During the course, the students faced several experiences on using the mutation testing criterion. In this context, one can notice the students were pretty experienced and some of them were already employed by software developing companies.

At the end of a complete course on Software Testing, we provided an eight-question survey aimed at assess what the students' feelings on using the mutation testing criterion as an activity to support the learning progress of novice programmers. The questions were focused on mutation testing practices and testing tool support for novice programmers. The complete survey, the possible answers, and the results are presented in Table III. The provided questions were objective and one can notice the possible answers and the percentage of the students' opinions about each issue through the column "Results". Section V provides further analysis and evidence collected from this questionnaire.

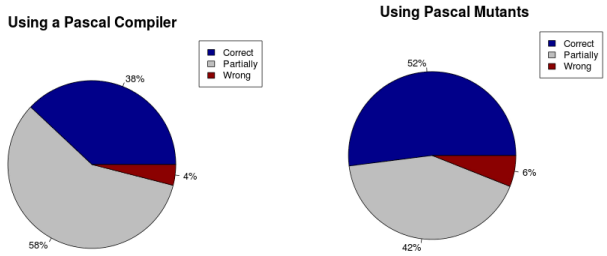
V. Results and Analysis

In this section we present the results achieved from the aforementioned analysis. In addition, we present possible answers for the pre-defined RQs by Analysis 1. Results and conclusions provided from the survey assessment are presented. Finally, we raise some key points about the practice of using mutation testing concepts to add knowledge to the learning process in programming courses.

A. Results to Analysis 1

Regarding Analysis 1, we evaluated each response applying the following scale-point: 3 points for complete/correct response, 2 points for incomplete/partially correct responses, and 1 point for incoherent/wrong responses. Then, each student received a score based on his performance during the experiment. Regarding all of the five survey questions (Table II), the students must have a score varying from five to 15 points. To avoid bias during this process, at least two of this paper's authors checked each question's responses. In cases of disagreements about the scores, a third author was appointed to grade the response. Further, we made all these verification ignoring group distinctions, once there was no information about the groups in the survey. Then, after defining the scores, each survey was associated to a student and, consequently, the groups were identified.

Figure 2 contains two fan plots representing an analysis of each group performance. This analysis highlights the quantity of questions considered correct, partially correct, or incoherent/wrong for each treatment. This evaluation considers all of the questions answered by each group. Summarily, students using *Pascal mutants* (G2) answered 52% of questions correctly (Figure 2b), while students in G1 only 38% (Figure 2a).



(a) G1: Regular Compiler. (b) G2: *Pascal mutants*

Fig. 2: Comparison between Groups' Performance.

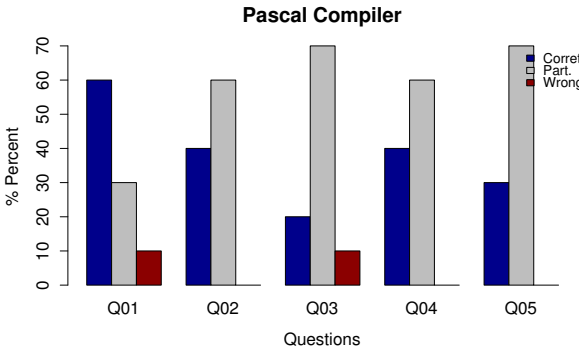
Along the same lines, regarding a broad evaluation on each question included in our experiment, Figure 3 presents a bar graph comparing the percentage of correct, partially correct and wrong answers for each question, in each group. Briefly, it is possible to notice a considerable improvement from G1 (Figure 3a) to G2 (Figure 3b), particularly in the number of correct answers. This shows that the students in G2 were better able to understand the code than those in G1.

Similarly, Table IV shows a bar graph with a comparison between the mean score for each of the five questions. For three of the five questions, the average of the students who explored the mutation tool was higher than the average of students who used a Pascal compiler. For one question (Q5), the average was exactly the same. And only for Q4 the students using a regular compiler obtained better scores.

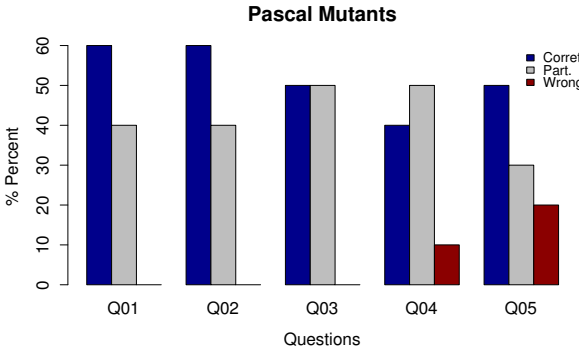
Another quantitative analysis we made considers the time it took each student to conclude the experiment. To realize this investigation, each student had specified a starting time and an ending time for their experiment. This analysis was performed

TABLE III: Analysis 2: Survey's Question.

#	Questions	Results
SQ1	What were your previous knowledge on software testing before the course?	None **42.86%** Intermediary **57.14%**
SQ2	Do you encourage mutation testing concepts to be presented in conjunction to programming foundations?	Yes **38.10%** Yes, superficially **52.38%** No **9.52%**
SQ3	What might be the effects of practicing mutation analysis by novice programmers?	Better programs **61.90%** Skilled better programmers **28.57%** None **9.52%**
SQ4	Do you consider regular testing tools useful to teach programming foundations?	No, they need to be alleviated **9.09%** Yes, using basic functionalities **54.55%** Yes **36.36%**
SQ5	Do you consider the mutation testing tools useful to teach programming foundations?	No, they need to be alleviated **61.09%** Yes, using basic functionalities **19.55%** Yes **19.4%**
SQ6	Disregarding the specific course on software testing, considering your course's curriculum, software testing concepts:	were not presented **27.27%** were fairly presented **13.64%** were poorly presented **59.09%**
SQ7	Do you think specific educational testing tools might be useful on creating good programming habits?	Yes **59.09%** No **41.1%**
SQ8	Do you think designing test cases through mutation testing might be useful to improve the learning capacity of novice programmers ?	Yes **72.73%** No **27.27%**



(a) Group 1: Regular Compiler.



(b) Group 2: *Pascal µutants*

Fig. 3: Percentage of Hits by Question.

TABLE IV: Mean Score for Each Question by Group.

Mean Score for Each Question					
Group	**Q01**	**Q02**	**Q03**	**Q04**	**Q05**
G1 – Regular Comp.	2.5	2.4	2.1	**2.4**	**2.3**
G2 – *Pascal µutants*	**2.6**	**2.6**	**2.5**	2.3	**2.3**

to help answering RQ 2, involving the trade-off analysis on applying the mutation testing criterion to teach programming. RQ 2 was actually designed to complement the results from RQ 1 including the statistics from the Analysis 2. Figure 4 represents three bar plots containing the total amount of time in minutes and the final score of each student separated by the groups they belong to. Students from G1 were quicker to complete the experiment than students from G2 (Figure 4a). Altogether, students in G2 took 17 minutes more than students in G1 to finish their experiments. On the other hand, students from G2 and obtained better scores (Figure 4b). Figure 4c presents a comparative analysis of the mean time and the mean score between the two groups.

In order to provide a specific comparison involving time and scores of the groups, we established a quantitative value named *Performance Factor* (PF). This factor is figured out through the score over the time in minutes. This number represents how effective the students were in considering their time and score. Table V presents the performance factor of some student divided by groups. G1 obtained an average PF of 0.40 and G2 obtained 0.25. Which means that the control group had a high PF and G2 spent a lot of time to complete their survey.

TABLE V: Performance Factors.

G1	.17	.22	.45	.36	.44	.45	.50	.43	**.40**
G2	.18	.23	.30	.25	.40	.40	.22	.17	**.24**

Through empirical analysis, effects of using the *Pascal µutants* tool and mutations testing to tech novice programmers are positive. It is possible to notice that stats and data collected from Analysis 1 revealed *Pascal µutants* positively affects the understanding about the source code. In general, students that have used *Pascal µutants* to conduct their activities obtained a better mean score in all of the questions with the exception of Q4. We noticed students in G1 tried to solve the survey's questions through trial and errors approaches. They explored the compiler with different inputs and observing the program behavior. On the contrary, students in G2 had to think more

carefully about the algorithm to "kill" mutants, consequently they were able to formulate more accurate answers about the program. Using the mutation analysis was a valuable experience for novice programmers.

We observed that students in G2 took a long time to finish their experiments. This reveals that the practical usage of a mutation testing tool instead of regular compilers may be considered a disadvantage. This is due to the fact that students in G1 had previously obtained skills to conduct the experiment using their preferred Pascal compiler. On the other hand, students in G2 had to dedicate more effort to understand all of the resources and functionalities provided by *Pascal mutants*

Regarding RQ1, we obtained valuable evidence that, for novice programmers, the mutation criterion can facilitate the learning process. The process of finding inputs to generate different outputs for the original program and mutants aids in the learning process. Further, the comparative analysis among similar source codes leads the student to play a tester role. In this sense, novice programmers have to perform successive evaluations and take decisions that contribute to the complete understanding of the semantic rules associated with the programming language. Then, the tool implemented and the experiences reported in this paper have helped students in understanding more deeply the concepts of the Pascal programming language. Numerically and visually, Figures 2 and 3 represent how efficient is the mutation testing to improve novice programmers' experiences.

Regarding RQ2, which was designed to define the trade-offs of our approach, we consider the experience reported in this paper reveals important findings. Through our experiment, the main trade-off we noticed was associated with the time to finish the activity. Students using a mutation tool will take a considerable time to figure inputs able to "kill" mutants. We noticed a need for specific strategies to make the usage of mutation testing faster, intuitive and efficient. Figure 4 presents some visual evidence on how efficient and cost effective is using *Pascal mutants*. During the experiment, students from G2 designed several manual outlines of the algorithm's behavior, contributing to the delay finishing their activity.

Still regarding the RQ2, during this research, we noticed several efforts and prerequisites that can be considered trade-offs of using a mutation criterion in the learning process of programming. Among these trade-offs we highlight three needs: (1) supporting material about mutation testing, (2) basic knowledge of software testing concepts, and (3) introductory lessons of software testing. However, we define the lack of adequate tools as the major limitation to a wide adoption of this approach. Regular tools for mutation testing are not indicated for novice programmers. In this context, to apply this testing criterion in programming courses, teachers have to adapt modules from regular mutation tools or even implement their own tools. In this context, whether Pascal is the programming language to be taught, we strongly advise teachers to use Pascal Mutants or any other adequate supporting tool.

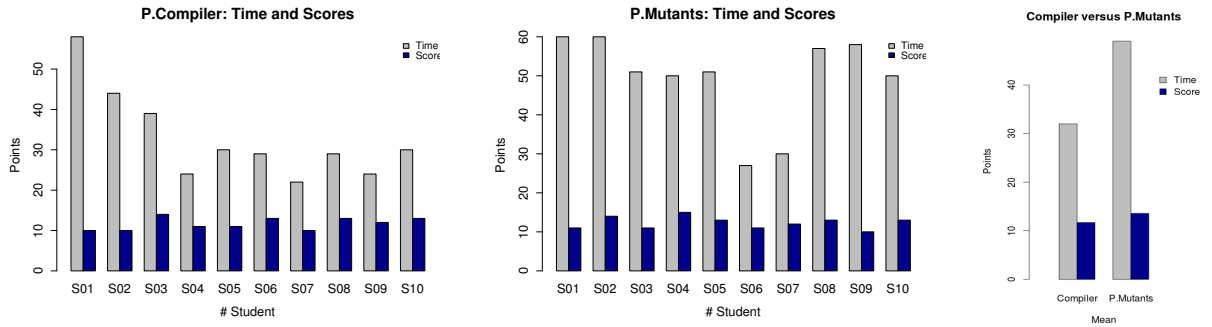
B. Results to Analysis 2

Regarding the survey assessment, whose results are presented in Table III, it is possible to notice that the senior students appreciate the idea of exploring mutation testing to complement the learning processes of novice programmers. Besides that, we noticed that before coursing a specific class focused on software testing, 42% of the students declared they had no previous knowledge on this topic (SQ1). That is a large percentage of concern that might directly effect the quality of the software developed by that student when he/she becomes a professional. It was notable the number of students (SQ2 – 90%) that agree with the practice of including mutation testing in programming courses. Besides several benefits such as, programmer skills, programs' quality, good programming habits, results collected from SQ3, RQ4, and SQ5 revealed that the students think it is necessary to have software testing tools specially designed to teach programming, as regular testing tools might be quite complex. 90% of the students think that practicing mutation testing contributes to better programmers (SQ3). RQ4 and RQ5 show that most of the students believe that regular testing tools are not recommended for novice programmers. This result is even worse for mutation testing tools, as 61% of the student believe the functionalities of regular mutation testing tools need to be alleviated. SQ6 and SQ7 lead us to consider the association of software testing notions to novice programmers: 87% of the students were not introduced to testing concepts and when introduced, these concepts were poorly explored (SQ6); 60% of the students see a strong relation between specific education testing tools and good programming habits. Finally, SQ8 reveals almost 73% of the students agree that the process of designing test cases to feed a mutation testing tools is a good exercise to improve the learning capacity of the student.

Further evidence can be collected through the quantitative analysis provided by Table III. In front of this evidence, we have two different conclusions: (1) mutation testing is one of the most appropriate testing criterion to help teaching programming in different languages; (2) there are no mutation testing tools specifically designed for novice programmers. In the context of the second conclusion, we can define the *Pascal mutants* states a significant contribution, being a tool specially designed to help novice programmers.

C. Discussion

The results raise some key points related to our own point of view about the usage of testing concepts in programming courses. According to our experience, the contact between novice programmers and testing activities may be profitable, establishing positive habits starting at the beginning of their professional career. For instance, regarding the questionnaire, we believe that cases in which the mutation approach resulted in minor or slight endorsements (S5 and S7) are due to the fact that regular mutation testing tools are not implemented to educational purposes. Then, some students recognize the importance of exploring mutation, however they highlight that efforts are necessary on using mutation testing tools properly.



(a) G1: Pascal Compiler.

(b) G2: *Pascal mutants*

(c) Mean Time and Score.

Fig. 4: Experiment: Times and Scores.

Most of the seniors believe mutation could be useful to complement their experiences when they were beginners, once testing tools specifically designed for educational purposes are explored. In addition, the students' contact with testing practices could contribute to revealing the importance of testing activities, providing the development of reliable applications.

Regarding Analysis 1, dividing the students into different groups may represent an effective environment for learning. In terms of code understanding, Table IV shows that mutation analysis leads the students to verify all of the code statements carefully, helping them to understand code issues more deeply. The score differences revealed by Table IV are small, however they are favorable to the group which has used mutation analysis to conduct the activity. The small differences are justified because the study conduct in this paper is considered small and the students were evaluated based on only 5 questions. Then, we still consider these results as significant. We believe this difference trends to be more representative with more deep studies, for example, a study comparing two different classes from different departments or universities in which only one group has previous knowledge on testing concepts. In addition, we believe the characteristics of our study (i.e., number of subjects, target systems, etc.), when contrasted with the state of the art in literature, represent a first step towards the generalization of the results achieved in this study.

In this scenario, additional ideas to explore similar environments may evolve into two scenarios: (1) simulating real teams of developers and testers, and (2) using a test case developed by one group in order to test systems developed by different groups. In this context, it is important to plan all of the activities before the beginning of the course because the mutation testing criterion will take more time than regular activities. We believe other similar activities may influence novice programmers to improve their skills and their knowledge about general program comprehension. It is important to conduct broader experiments to obtain a more in-depth validation of the ideas presented in this paper. A possible future experiment may involve the observation of two different groups of students during a semester-long course. Another possibility is to reproduce the same experiment using different programming languages and compare the results.

D. Threats to Validity

An *internal* threat to the validity of our study concerns the better results earned by G2, once they spent more time analyzing the source code during the mutation analysis. Furthermore, an *external* validity to the study is the fact that the results observed in our analysis are provided by a small-scale study. Regarding the survey analysis presented in Table IV, we noticed a threat associated with its *conclusion* validity, since senior students may be too far from first-year students to really remember what it was like, leading them to misinterpret the survey. We consider mutation testing a powerful resource to catch students' attention and improve their learning processes, however students have to face continued experience on mutation testing to make their learning process more valuable.

VI. Conclusion

This paper advances a preliminary study on evaluating the mutation testing criterion as to improve the learning processes of novice students in programming courses. We present a two-fold analysis (an practical analysis and a survey assessment) on the effects of using mutation testing to improve learning processes in programming courses. As methodology, we developed and provided an open-source mutation testing tool. *Pascal mutants* is specially designed for education purposes. Results reported in this paper are enough to reveal that mutation testing criterion is profitable for programming courses. Two main contributions are associated with this study: (1) the analysis states that mutation testing can be seen as a promising testing criterion to support teaching programming foundations to novice students; (2) this study provides a more stable version of *Pascal mutants* which supports mutation testing to the educational purpose of teaching programming for novices students. We consider this second contribution as relevant, once there is a notable lack of testing tools for educational purposes. Also, we consider the usage of software testing concepts as a profitable strategy to teach novice programmers, composing the main implication of our research.

REFERENCES

[1] M. A. S. Brito, J. A. L. Rossi, S. R. S. Souza, and R. T. V. Braga, "An experience on applying software testing for teaching introductory programming courses," *CLEI Electronic Journal*, vol. 15, pp. 1–5, 04 2012.

[2] M. Butler and M. Morgan, "Learning challenges faced by novice programming students studying high level and low feedback concepts," in *Proc. of Ascilite 2007*, Singapure, 2007, pp. 99–107.

[3] R. M. Kaplan, "Teaching novice programmers programming wisdom," in *Proc. of PPIG 2010*, Madrid, Spain, 2010, pp. 1–9.

[4] K. Buffardi and S. H. Edwards, "Effective and ineffective software testing behaviors by novice programmers," in *Proc. of ICER 2013*, New York, USA, 2013, pp. 83–90.

[5] D. Souza, B. Oliveira, J. Maldonado, S. Souza, and E. Barbosa, "Towards the use of an automatic assessment system in the teaching of software testing," in *Proc. of FIE 2014*, 2014, pp. 1–8.

[6] A. Bertolino, "Software testing research and practice," in *Abstract State Machines 2003*, ser. Lecture Notes in Computer Science, E. Brger, A. Gargantini, and E. Riccobene, Eds. Springer Berlin Heidelberg, 2003, vol. 2589, pp. 1–21.

[7] E. Jones, "An experiential approach to incorporating software testing into the computer science curriculum," in *Proc. of FIE 2001*, vol. 2, 2001, pp. F3D–7–F3D–11 vol.2.

[8] R. A. P. Oliveira, L. B. R. Oliveira, B. B. P. Cafeo, and V. H. Durelli, "On using mutation testing for teaching programming to novice programmers," in *Proc. of ICCE 2014*, Nara, Japan, 2014, pp. 394–396.

[9] R. Demillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[10] D. Souza, J. Maldonado, and E. Barbosa, "Progtest: An environment for the submission and evaluation of programming assignments based on testing activities," in *Proc. of CSEE T 2011*, May 2011, pp. 1–10.