

All Uses and Statement Coverage: A Controlled Experiment

Diego Vallespir¹, Silvana Moreno¹, Carmen Bogado¹, Juliana Herbert²

¹Universidad de la República, School of Engineering, Montevideo,
Uruguay

²Herbert Consulting, Porto Alegre,
Brazil

{dvallesp, smoreno}@fing.edu.uy, cmbogado@gmail.com, juliana@herbertconsulting.com

Abstract. This article presents a controlled experiment that compares the behavior of the testing techniques Statement Coverage and All Uses. The design of this experiment is typical for a factor with two alternatives. A total of 14 subjects carry out tests on a single program. The results indicate that there is enough statistical evidence to state that the cost of executing All Uses is higher than that of executing Statement Coverage – a result that we expected to find. However, no statistical differences were found as regards the effectiveness of the techniques.

Keywords. Empirical software engineering, testing techniques, test effectiveness, test cost.

1 Introduction

Software unit testing is strongly established in industry. However, the effectiveness and cost of each different unit testing technique is not known with certainty. This makes the decision of which technique to use hardly trivial.

Many years of empirical research have gone by and yet final results have not been achieved. *In A look at 25 years of data* the authors examine in depth different experiments of software testing reaching the same conclusion [12].

A series of formal experiments are currently being carried out at the Computer Science Institute of the School of Engineering of the Universidad de la República in Uruguay in order to gather more precise data in this direction. Four experiments have finished at present and this article describes

one of them. The results of other experiments of this series are included in [15, 16, 17].

Defect-detection techniques are the mostly used means to achieve quality in software. Therefore, we need to understand the relations between costs and benefits regarding those techniques, especially when aiming to compare them. The experiment hereby described compares the unit testing techniques All Uses (AU) and Statement Coverage (SC) with aims to know its cost and its effectiveness. The cost is defined as the time it takes to develop the test cases in order to comply with the coverage demanded by the technique. Effectiveness is defined as the number of defects encountered when executing the technique divided by the number of total defects of the program being tested.

A group of undergraduate students of Computer Science of the Universidad de la República in Uruguay develop test cases using SC and AU to execute a program written in Java. These students record the defects found as well as the time employed in the development of the test cases. We analyzed the effectiveness and the cost of each subject and we suggested hypothesis tests to find out whether among the techniques used there are differences of effectiveness and/or cost.

The rest of the article is organized in the following way: section 2 describes the techniques that were used in the experiment. Section 3 presents the related work. The experiment set up is presented in section 4. The results of the experiment

are presented in section 5 and the discussion in section 6. The most important threats to validity are presented in section 7. The conclusions and future work are presented in section 8.

2 Background: Statement Coverage and All Uses

Two testing techniques, both white-box, are employed. SC that is based on control flow and AU that is based on data flow.

In order to satisfy the prescription of SC technique each statement of the source code must be executed at least once when running the tests. Since this technique is widely known, we do not go deeper into it in this article.

The AU technique expresses the coverage of testing in terms of the definition-use associations of the program. A *definition* of a variable occurs when a value is stored in the variable ($x := 7$). A *use* of a variable occurs when it is read (or uses) the value of that variable. This can be either a p-use or a c-use. A *p-use* is the use of a variable in a bifurcation of the code (*if* ($x == 7$)). A *c-use* is when the use is not in a bifurcation. For example, in ($y := 7 + x$) there is a c-use of x (there is also a definition of y).

The control flow graph is a representation through a graph of the different execution paths that can be taken by a program. The nodes of the graph represent the statements (or code blocks) and the edges the bifurcations (*if*, *for*, *while*, etc.). We will use i_j to refer to a particular node in the graph.

Then, an execution path of the program can be represented as a sequence of nodes. For example, i_1, i_4, i_7 represent an execution path where node i_1 is executed first, then node i_4 and finally node i_7 .

A definition of a variable x in a node i_d *achieves* a use of the same variable in a node i_u , if there is a definition clear path from i_d to i_u in the control flow graph and the path is executable. A path $i_1, i_2, \dots, i_n$ is a *definition clear path* for a variable x if the variable x is not defined in the intermediate nodes of the path ($i_2, \dots, i_{n-1}$).

AU requires that at least one definition clear path be executed from each definition (of every variable) to each achievable use (of the same variable).

The classical definitions of the techniques based on data flow and particularly AU are presented in an article by Rapss and Weyuker [14].

In Object Oriented languages the basic testing unit is the class. It is necessary to test its methods in an individual and in a collective way, so as to test the interactions generated through the sequence of calls originated by the invocation of a particular method. AU can be applied both for the tests of individual method belonging to a class and for the methods that interact with other methods of the same class or of other classes.

The tests of a class in AU can be carried out in two levels: Intra-method (Intra) and Inter-method (Inter). In **Intra**, only the method under test is considered for the code coverage. Therefore, in this case, the methods that interact with the method under test are not considered at the moment of developing the test cases. On the other hand, in **Inter**, the methods that interact with the method under test are considered for the code coverage too.

Two types of definition-use pairs to be tested are identified in relation to the levels presented previously. The **Intra-method Pairs** are those which take place in individual methods and test the data flow limited to such methods. Both definition and use belong to the method under test.

The **Inter-method Pairs** occur when there is interaction between methods. They are pairs where the definition belongs to a method and the corresponding use is located in another method that belongs to the chain of invocations.

In most of the literature that presents techniques based on data flow, the examples that are given contain simple variables such as integers and Booleans. However, criteria that normally are not treated should be defined at the moment of applying these techniques in arrays or even more difficult in objects.

Establishing these criteria is essential in order to know under which conditions the technique is applied. Different conditions can produce different results in the effectiveness and cost of AU since in fact, they are different techniques with the same

name. Many of these conditions refer to how the Inter-method Pairs should be considered. This experiment establishes the conditions for the application of the AU technique based on what is proposed in [5, 6, 7].

3 Related Work

Several formal experiments were carried out in order to find out the effectiveness and/or cost of different unit testing techniques. Some experiments that use techniques based on data flow are presented in this section.

In 2011, Kakarla, Momotaz and Namin presents a meta-analytical approach to quantify and compare mutation and data flow testing techniques based on findings reported in research articles [9]. Previous investigations of Mathur and Wong [11], Offutt [13] and Frankl [4] were selected and mutation and data flow testing techniques are compared. Selected papers programs was developed in C with 40, 18 and 39 LOCS respectively. The Data flow techniques used were All-Uses, pairs DU and All-Uses respectively. The study assesses the effectiveness and efficiency of techniques. The efficiency refers to the number of test cases required to achieve an adequate test suite with respect to mutation or data-flow coverage criteria, whereas effectiveness refers to the proportion of the faults that was revealed using each testing technique. The results show that mutation is at least two times more effective than data-flow testing. The data-flow testing outperforms, by three times, mutation testing in terms of efficiency

In 2006, Andrews and others carry out an experiment to compare the techniques: Block, C-Use, P-Use and Decision [1]. The program used was developed in C with 5905 NLCS and the defects were injected through mutants generation. The test suites were generated randomly for various coverage criteria and levels, with the aim of obtaining a spread of test suites that span all coverage levels in a balanced manner. The cost was measured as the number of test cases necessary to achieve the coverage, and the effectiveness was measured as the number of defects detected. The results indicate that the level of less expensive coverage is Block, C-Use, Decision and P-Use, in that order.

Regarding the effectiveness, C-Use and P-Use are better in detecting defects in relation to Decision and Block.

In 1990 Weyuker presents an experiment in order to find out the cost of the testing techniques based on data flow [18]. The cost is measured as the number of test cases developed when applying the technique. The following testing techniques are studied: All c-uses, All p-uses, All Uses and All Paths Definition-Use. The results show that the number of necessary test cases to satisfy those criteria is much lower than the level of the worse case calculated theoretically on a previous work also by Weyuker [19].

Frank and Weiss present an empirical study in which they compare the effectiveness of the All Uses and Decision Coverage techniques [3]. Nine programs are used and random test cases are developed for each of them. No human testers take part in this experiment. Sets of test cases, that meet one or the other criterion, are taken and whether each of these groups detects at least one defect is evaluated. The results show, with 99% confidence, that the All Uses criterion is more effective in 5 out of the 9 programs. In the other 4 programs it is impossible to differentiate.

In 1994, Hutchins and others published an experiment the goal of which is to compare the effectiveness of a variant of the technique All Paths Definition-Use and a variant of the Decision Coverage technique [8]. The experiment has similar characteristics to that of Frank and Weiss. However, in this experiment both test cases automatically generated at random and human testers are used. The results show that there is no statistical evidence indicating that one technique is more effective than the other.

Li and others carry out an experiment to compare four unit testing techniques: Mutants, All Uses, Edge-pair Coverage and Prime Path Coverage [10]. They try to find out the effectiveness (measured as the number of defects detected on the seeded defects) and the cost (measured as the number of test cases it is necessary to develop in order to meet each testing criterion) The cases were developed by hand with the help of tools to know the coverage and another one to generate mutants. The result is that the Mutant technique

finds more defects while the other three are similar. Surprisingly (according to the authors) the Mutant technique was the one that required the least test cases.

One of the points that we consider weak in these experiments is that they measure the cost as the number of test cases it is necessary to develop in order to satisfy a certain testing criterion. We believe that the time employed in developing these cases is a more interesting measure for the cost.

4 Experiment Setup

The design of the experiment and its execution are presented in this section.

4.1 Goals, Hypotheses and Metrics

The aim of our experiment is to evaluate and compare the SC and AU techniques concerning their effectiveness and cost. To document our goals, hypothesis and variables we use the GQM approach [2].

Analyze Statement Coverage and All Uses techniques
for the purpose of their evaluation
with respect to their effectiveness and cost
from the viewpoint of the researcher
in the context of a course thought and done especially for this experiment in the School of Engineering, Universidad de la República of Uruguay.

Since the generic objective is clearly divided in two (effectiveness and cost) we propose different objectives for each of these options. The viewpoint and the context do not change as regards the general objective.

Goal 1:
 Analyze Statement Coverage and All Uses techniques
 for the purpose of their evaluation
 with respect to their effectiveness. . .

Goal 2:
 Analyze Statement Coverage and All Uses

techniques
 for the purpose of their evaluation
 with respect to their cost. . .

The model to evaluate the effectiveness of each individual is defined as the number of defects found by that individual divided by the number of total defects of the program under test. This model is show in Fig. 1

The model to evaluate the cost of executing a technique is defined as the time in minutes it takes to design and codify the test cases using the technique.

The questions, metrics and hypotheses associated with each goal are the following:

Goal 1:

Q1. Which is the average effectiveness of the individuals when executing a technique?

H1. The individuals that apply All Uses have a better performance than those who apply Statement Coverage as regards the average effectiveness.

M1.1. Number of defects found by each subject.

M1.2. Total number of defects in the program under test.

Goal 2:

Q1. Which is the average cost obtained by individuals when executing a technique?

H1. The individuals that apply All Uses obtain a higher cost than those who apply Statement Coverage as regards the average of the cost.

M2. Total time of design and codification of test cases by each subject.

The **experimental unit** of the experiment is a program written in Java language. Its main feature is payroll accounting. The program is presented in subsection 4.5.

The **factor** is the testing technique. The **alternatives** of this factor are the techniques to be evaluated: SC and AU.

The **response variables** considered in this experiment are the effectiveness and the cost of the techniques.

$$\text{Effectiveness of the individual} = \frac{\text{Number of defects found}}{\text{Total number of defects of the program}} \quad (1)$$

Fig. 1. Effectiveness of each individual definition

The **hypotheses** for this experiment are also the traditional in this kind of experiment. The null hypothesis of effectiveness, hypothesis that we want to reject, states that the medians of effectiveness of the techniques are the same. The null hypothesis of cost states that the medians of cost of the techniques is the same. The alternative corresponding hypothesis simply indicates the medians are different.

4.2 Subjects

The subjects of the experiment are graduate students of Computer Science of the Universidad de la República of Uruguay. All of them are advanced students since they are coursing fourth or fifth year. They have passed the course “Programming Workshop” in which Java language is learned and they have completed at least another three courses in Programming and a course in Object Oriented Programming. All of them have completed a Software Engineering course in which, among other things, different testing techniques are studied. We consider, therefore, that the group that participates in the experiment is homogeneous due to the fact that they are at the same level in their studies and that they have been provided with the same training as part of the experiment.

The students participate in the experiment in order to get credits for their studies and this is their motivation. The formal framework is a course in the degree of Computer Science designed especially for the experiment. Attendance to the training is mandatory as well as the execution of the testing technique following the material provided by the researchers. They know that completing the course successfully depends on following the script provided to apply the technique correctly and not on the number of defects found.

The students are not aware that they are taking part in an experiment; they believe they are taking

a course with an important component of laboratory practice.

It is the students who enroll in the course. This course is not mandatory for the degree they are taking, consequently their participation is voluntary.

4.3 Experimental Materials

The experiment material consists of the Class Diagram of the program under test, the program user documentation, the program’s Javadoc, an electronic spreadsheet to record defects and times, a script to conduct the testing activity and the source code of the program. The script and the program are presented in independent sections.

The java classes of the program must be tested following the bottom-up approach during the experiment. The Class Diagram serves this purpose and it is given to the subjects with a list that specifies the order in which each class has to be tested.

While the test is being conducted the subjects record the defects they find in the program as well as the design and JUnit codification time of the test cases. These data is recorded in the spreadsheet for recording defects and time that was created for this purpose.

The spreadsheet contains two sheets, the first is presented in Fig. 2. The name of the subject, the name of the program under test, the testing technique being applied, the starting and finishing date, and the design and codification times of the test cases of each Java class are recorded here.

In the second sheet, a new row is entered for every defect that is found. The number of the code line where the defect is and its description is recorded for each defect found, as well as the name of the class that contains it. Finally, it is specified whether the defect was found during the design (design and codification of cases) or during the execution of the test cases.



Defects Recording System				
Name		Technique		Software
User		Starting date		Finishing time
		Design times		Main execution time
		TypesPosition.java		
		TypesDiscount.java		
		Bands.java		
		Discount.java		
		MaxMinCode.java		
		Position.java		
		Discount_Receipt.java		
		Payslip.java		
		Employee.java		
		Logic.java		
		Persistence.java		
		Main		

Fig. 2. Defects and times recording spreadsheet - Sheet 1

4.4 Testing Script

The subjects use a script to conduct the tests during the experiment. This script is presented and explained to the subjects during the training provided as part of the experiment.

The script is the process that must be followed by the subjects. It is made up of three phases: Preparation, Design and Execution. The complete script can be found in Appendix A.

During the Preparation phase the subject must carry out an initial check-up that guarantees that the tests can be conducted. He must check that he has the source files to test, a file with the class diagram of the program under test, the Javadoc of the classes and the spreadsheet to record defects and times that he will use during the tests.

During the Design phase the subject develops the test cases following the prescriptions of the testing technique previously assigned. The subject must design the test cases following a bottom-up approach. After having designed the test cases, the subject codifies them in JUnit. In this phase, the time it took to design and codify the test cases is recorded in the spreadsheet.

The execution of the test cases is conducted during the Execution phase. The cases must also be executed using a bottom-up approach. The phase ends when there are no test cases that produce a failure. While there are test cases that produce failures the subject must:

- Choose one of the cases that produce a failure.
- Find the defect in the program that produces the failure.
- Record the required data in the spreadsheet for each defect found.
- Request the correction of the defect to the research team.
- Run the test case again to confirm that the correction has been made properly by the research team.

4.5 The Program

The program used in this experiment was built especially for an experiment conducted in 2008 [15]. This program was developed by a fourth-year student of Computer Science. The student developed the program using a specification written by the research team. This specification describes a program to calculate the salaries for educational staff and non-educational employees in a fictitious university.

The functionalities of the program are basic as regards the calculation of salaries. It allows recording employees and their positions. It offers the functionality of reallocating positions, but an employee cannot hold more than one position (be it educational or non-educational). It makes it possible to raise salaries in different ways. It also has

a functionality of making statements in which the calculations of all the employees of the system are made generating their corresponding salary slips. It also provides the possibility of creating the salary slip for an employee in particular.

The student must develop the program in Java language taking the specification as a starting point, making sure only that the program could be compiled. That is to say, the student was not allowed to conduct any testing on the developed program. He was also asked not to conduct any type of static review. Therefore the delivered program only had corrections to the defects detected during the compilation.

Therefore, the program contains real defects that were not seeded in the code by the researchers. Besides, since the developer is not allowed to conduct any type of verification, the program is tested for the first time during the experiment, situation which we wanted to simulate.

The student that builds the program develops its documentation using Javadoc. He hands in installation and configuration manuals and a user manual that covers all the functionalities of the program.

The program has a size of 1820 LOCs (without comments) distributed into 14 classes. It is made up of 5 DataTypes, a class that implements the persistence of the data, 5 classes that contain the logic, 2 interfaces and a main class that implements the interface with the user.

The program has a small database (that use the HSQLDB database manager) to support its functionalities. This database is made up by 8 tables. The subjects are given a script for the creation of the tables and load of the basic data.

The defects of the program are not totally known since they are not seeded defects, but they correspond to those that appear during the development. During the 2008 experiment and during the experiment presented in this article, several subjects conducted tests on the program with different techniques. We consider that the number of defects that result from the union of the detected defects in both experiments is a good estimate of the total defects of the program. This number is 187.

4.6 Experiment Design

The design of an experiment corresponds to the design of a factor (testing technique) with two alternatives (SC and AU). The subjects that participated in the execution of the experiment are a subgroup of the subjects that participated in the training; 4 from the first training and 10 from the second. Out of these 14 subjects, 8 apply AU and 6 apply SC.

The subjects in the training could choose freely to participate in the experiment in order to get more academic credits. The final design is not balanced due to the fact that each subject carried out the practice only with one technique during the training.

All the subjects use only one technique (the allotted one: CS or AU) on an only program (the Accounting program).

4.7 Training

The subjects who enroll in the course must go through the training that aims at ensuring that they have the necessary knowledge and practice to use the SC and AU techniques correctly. The training is made up of three parts: JUnit Learning session, Techniques Learning session and Execution session. Fig. 3 presents the different activities in each session.

The aim of the JUnit Learning session is for each subject to learn how to use the JUnit tool that they will be using to codify the test cases. Each subject is given a simple program specification and they are asked to implement that specification in Java and to develop a JUnit class to test its functionality. The idea is that the subject should study JUnit individually since this tool is not explained during the training.

This session is done at home by the students and takes a week. Once it finishes, the students hand in the Java class and the JUnit class with the codified test cases. These are reviewed by the researchers in order to verify that the student has acquired the necessary knowledge with JUnit.

Once the JUnit Learning session is finished the Techniques Learning takes place. This session has the aim of allowing the subjects to learn the SC and AU techniques. A theoretical/practical course of 9 hours is conducted during a day. A theoretical

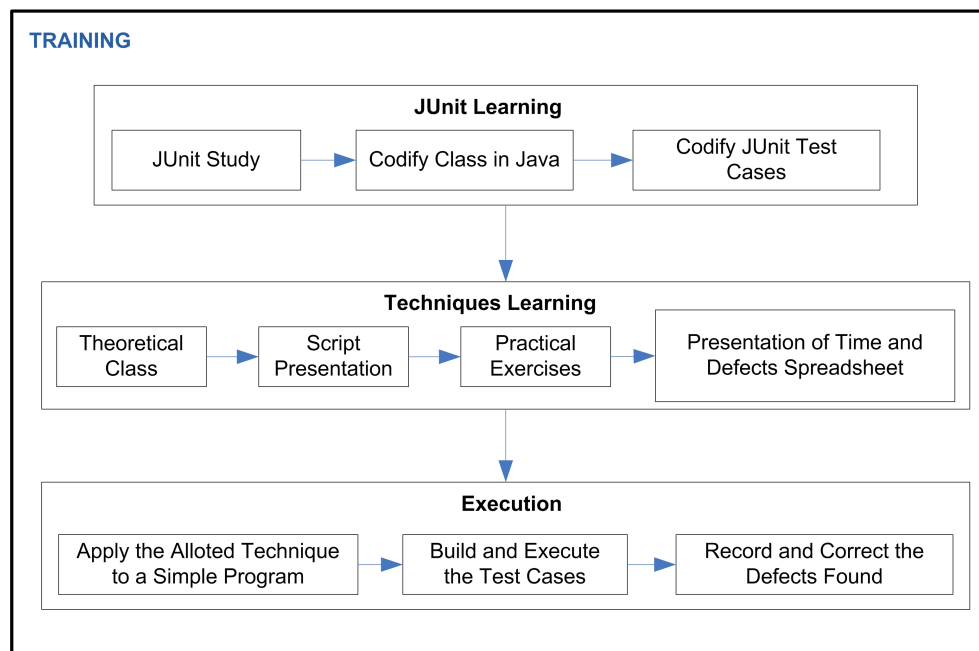


Fig. 3. Training of the subjects

class to explain the testing techniques to be used is given in the first half of the day. The verification script to be followed when conducting the tests and the spreadsheet to register times and defects are also explained. During the second half of the day practical exercises to be solved in groups or individually are done. These exercises are an intense practice of the techniques of the experiment.

The Execution session is also conducted during a whole day, seven days apart from the training session. In this session the subjects individually apply the technique allotted to each one, developing the necessary test cases and executing them. This training session is considered also as an experiment in itself [16].

In order to complete the work they follow the script provided in the Techniques Learning session, registering the time and the defects as indicated in the same. The students who are unable to finish their task during the day have a week to finish it. At the end of the session the subjects hand in the JUnit classes developed, the spreadsheet with the registers of time and defects and the notes they have made in order to be able to apply the

technique (control flow graphs, identified paths, etc.) Once all the deliveries have been done, the research team reviews them and they are given back individually to each student.

The execution session serves to achieve several objectives. The subjects familiarize with the verification script they must follow, with the techniques and with the spreadsheet to use to keep the records. While the work is reviewed, it is possible to make adjustments and correct the mistakes that the subjects could be making while applying the technique.

4.8 Operation

The complete training takes place twice with two different groups of subjects. The first training and the second were carried out some weeks apart from one another.

10 subjects participated in the first training. 5 used AU technique and the other 5 applied SC. 11 subjects participated in the second training. 6 used AU and 5 used SC. The choice of subjects for the different techniques is at random. Each

complete training (the three sessions) took about three weeks.

The experiment with the Accounting program is conducted in 8 one-week sessions. Each session is allotted a certain number of Java classes to be tested by the subjects. The choice of classes to be tested each week is made based on the bottom-up approach and their complexity, and was made by the research team.

The execution of the experiment by the subjects is done at their home. At the beginning of each week the classes to be tested are sent to them via e-mail.

During the week the subjects report the defects found to the researchers requesting their correction. The research team sends the corresponding correction for each defect. The subjects are not allowed to make the corrections by themselves. The purpose of this is that the subjects have the same correction for the same defects.

It is important to point out that we have no knowledge of other experiments conducted in which the defects are corrected. The correction of defects during the experiment simulates better the testing activity in the industry.

Each weekly delivery made by a subject is validated by the research team. The aim is to make sure that the spreadsheet is complete. It is also controlled that the test cases handed in codified in JUnit do not fail, which means that all the defects that produce failure have been corrected.

A form that records the defects of each Java class of the program is completed using the defects found by the different subjects. It is made taking the defects form of the 2008 experiment as a starting point. This form is used to keep all the defects found by all the subjects per class of the program.

5 Results

The results are presented in two parts. The first sub-section considers goal 1: effectiveness of the techniques. The second considers goal 2: cost of the techniques.

Table 1 presents the subjects that used each one of the techniques, the Effectiveness (%) and the Cost (hours).

Table 1. Effectiveness (%) and Cost (hours)

Tech.	Subj.	Effect. (%)	Cost (hs.)
SC	S1	42%	54.6
SC	S2	15%	30.9
SC	S3	5%	41.3
SC	S4	6%	41.1
SC	S5	15%	35.8
SC	S6	7%	35.1
AU	S7	35%	53.5
AU	S8	3.7%	31.3
AU	S9	5%	45.1
AU	S10	15%	54.6
AU	S11	5%	29.5
AU	S12	23%	68.7
AU	S13	21%	105.2
AU	S14	4%	92.0

5.1 Effectiveness

In this section we present the descriptive statistics and the hypothesis test for the effectiveness of the techniques.

Descriptive Statistics

Fig. 4 presents a box and whisker chart of the effectiveness both of the SC technique and the AU technique. The medians of both techniques are similar, for SC it is 11% and for AU it is 10%. The distribution of both techniques is a bit different.

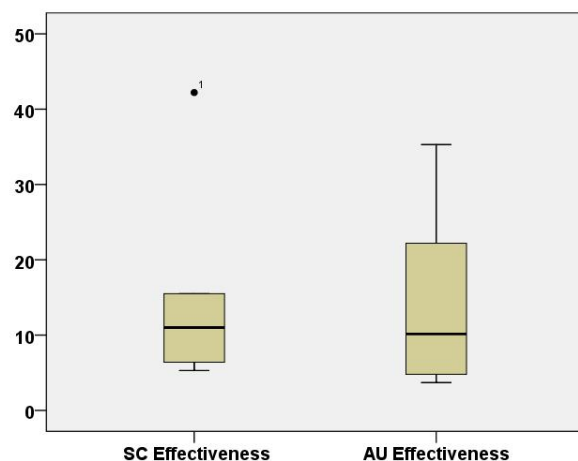


Fig. 4. Box and Whisker of the Effectiveness

Table 2 presents the number of subjects that used each one of the techniques, the median and the interquartile range calculator.

Table 2. Median and Interquartile Range Calculator

	No. of Subjects	Median	IRQ
SC	6	11%	9%
AU	8	10%	17.5%

Hypothesis Testing

The hypothesis test done to compare the effectiveness of the techniques is presented below. The quantity of observations we have (6 and 8) are too few to make parametric tests. Therefore, the Mann-Whitney test is applied, since it is suitable for our experiment design. The null hypothesis that indicates that the medians of effectiveness of both techniques are the same is proposed, together with the corresponding alternative hypothesis:

$$H_0 : Mdn_{ESC} = Mdn_{EAU}$$

$$H_1 : Mdn_{ESC} <> Mdn_{EAU}$$

The test of Mann-Whitney indicates that it is not possible to reject the null hypothesis. Therefore, we do not find there is a statistical difference between the effectiveness of both techniques.

5.2 Cost

In this section we present the descriptive statistics and the hypothesis test for the cost of the techniques.

Descriptive Statistics

Fig. 5 presents a box and a whisker chart of the SC and the AU cost. The median of SC is 38.5 hours and that of AU is 54.1 hours. The distribution of both techniques is very different. Although the minimum values are similar, the percentiles from 25% to 75% are totally different.

Table 3 presents the number of subjects that used each one of the techniques, the median and the interquartile range calculator of the cost (in hours).

Hypothesis Testing

The Mann Whitney test is used again. The null hypothesis in this case is that the median of the cost of the techniques is the same:

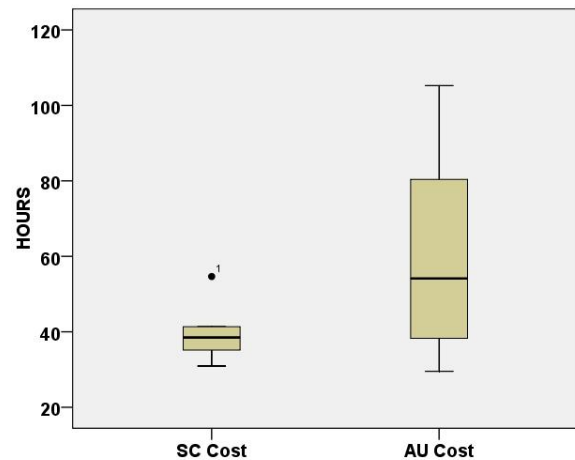


Fig. 5. Box and Whisker of the Cost

Table 3. Median and Interquartile Range Calculator of the Cost in Hours

	No. of Subjects	Median	IRQ
SC	6	38.5	6.18
AU	8	54.1	42.12

$$H_0 : Mdn_{CSC} = Mdn_{CAU}$$

$$H_1 : Mdn_{CSC} <> Mdn_{CAU}$$

The calculated p value is lower than the α chosen level (0.1), therefore we conclude that there is statistical evidence that indicates that AU is more expensive than SC.

6 Discussion

In this experiment we expected to find that the effectiveness of the AU technique is higher than that of SC. However, this could not be proved statistically.

We found that both techniques had a very low effectiveness, the median of SC being 11% and that of AU 10%. This effectiveness might be because the students did not know how to apply the techniques well. This can become more negative for the more complex technique, in this case AU. We are currently carrying out a study to know, taking the test cases developed by the subjects as a starting point, how far the subjects satisfy the prescription of the technique they use. This

study can help discuss the low effectiveness of both techniques in our experiment.

On the other hand, as far as the cost analysis is concerned, we found what we expected to find. The AU technique is more expensive than the SC technique. As we consider the design and codification time of the test cases as the cost of the technique, we can conclude that the AU technique is more complex to apply than SC.

The AU technique presents a greater variability in the cost than the SC technique. It could be thought from a practical point of view that the costs of applying SC are kept more under control than the costs of applying AU.

However, this does not have to be so. Our ongoing study that studies the compliance with the prescription of the technique by the subjects indicates that those who apply SC normally comply with it while those who apply AU have a great variability. This could explain the variability of costs; those who apply AU correctly could be those subjects who have a high application cost. These are assumptions under analysis at the moment.

7 Threats to Validity

There are various threats to the validity of this experiment. These make it important to replicate the experiment in order to know if the conclusions can be generalized. Some of the threats that we consider most important are mentioned below.

Only 14 subjects participated in the experiment. Thus, we can only use nonparametric tests. It is necessary to replicate the experiment with more subjects in order to generalize the findings.

The subjects are all undergraduate students. Although they are all advanced students and they are carefully trained, they are not professionals in software development. This might imply that the test cases developed could be “worse” than those an expert could develop.

Only one program is used in the experiment. Therefore, the obtained results may be due to the program’s special features and not to the techniques studied. It is necessary to replicate the experiment with different types of programs in order to generalize the findings.

The obtained results are only useful for SC and AU. Although in this experiment both techniques show very low effectiveness (about 10%), this cannot be generalized to other white-box techniques. Future replications must consider other white-box techniques and include black-box techniques.

8 Conclusions and Future Work

This article presents an empirical study that aims to find out more about the effectiveness and cost of the SC and AU techniques. As to the effectiveness, we could not reject the null hypothesis; therefore we cannot say that one technique is more effective than the other.

As far as the cost is concerned, although the subjects taking part in the experiment were few, we can say that the AU technique is more expensive than the SC technique. It is on average 50% more expensive to execute.

It is necessary to make replications in order to generalize the findings due to the different threats to the validity of this experiment. In fact, it would be interesting to increase the number of subjects, vary the testing techniques and have different programs to test.

If future replications of this experiment show the same result, we could say it is convenient, concerning the cost-benefit relation of executing white-box techniques, to use simple white-box techniques. The use of complex techniques such as AU would not be worth while, since their cost is much higher and there are no differences as to effectiveness.

As far as future work is concerned, it is our intention to replicate this experiment with the same type of subjects but a bigger number of them. This would eliminate one of the threats to the validity that were mentioned. In a longer period we would have more programs to test and finally we would add other testing techniques.

A The Script

The script for white box testing used in the experiment is presented below. The script describes the process to conduct the tests by the subjects. It entails three phases: Preparation, Design and Execution.

A.1 Script for white box testing**Step 1. Preparation:**

- Preparation activities for testing.

Step 2. Design:

- Design the test cases that satisfy the prescription of the technique.
- Build the test cases in JUnit.
- Record the required data of the phase.

Step 3. Execution:

- Execute the designed test cases.
- Find the defects associated to the cases that produce failures.
- Record the required data of the phase.

A.2 Script for the Preparation Phase

Preparation phase: Carry out an initial check-out to guarantee that verification can be done.

Step 1. Verify Files:

- Verify that the source files are available.
- Verify that you have the Class Diagram.
- Verify that you have the Javadoc of the classes to test.
- Verify that you have the defect and times spreadsheet.

A.3 Script for the Design Phase

Design Phase: Make the design of the test cases satisfying the prescription of the testing technique to apply. The cases must be designed according to the bottom-up approach.

Step 1. Define the data test set:

- Record the starting time of the activity.
- Define for each method the set of input values that satisfy the prescription of the technique.

Step 2. Define the expected results:

- Define the expected result (or expected behavior) for each element of the set of input values and thus make the test cases.

Step 3. Debugging:

- Eliminate the test cases that cannot be executed (impossible paths, etc.). Check if the prescription is still satisfied. In case it is not go back to step 1 trying to satisfy it.

Step 4. Codification of test cases in JUnit:

- Codify all the designed test cases in JUnit.

Step 5. Record of finishing:

- Record the design finishing time.

A.4 Script for the Execution Phase

Execution Phase: Carry out the execution of the designed test cases. The cases must be executed following the bottom-up approach.

Step 1. Execute the test cases:

- Execute the test cases.

Step 2. Analyze the obtained output:

- If there was no failure the phase is finished.

Step 3. Find defects:

- While there are test cases that produce a failure:
 - Choose one of the cases that produce a failure.
 - Find the defect in the program that produces the failure.
 - Execute step 4.
 - Request the correction of the defect to the research team.
 - Run the test case again to confirm that the correction has been made properly by the research team.

Step 4. Defect recording:

- For each defect found record the following data:
 - General description of the defect. It is important that the description is clear and accurate.
 - Name of the file that contains the defect.
 - Line in which the defect is. In case the defect is not on a specific line record 0 (zero). Beware: if the file that is being tested has been modified compared to the original (because some defect has been corrected), the line of the original file must be indicated.
 - Structure associated to the defect (i.e. IF, FOR, WHILE, name of the method, etc). If the defect has no associated line this field must be completed necessarily.
 - Starting line of the structure (mandatory if an associated structure is indicated).

References

1. Andrews, J. H., Briand, L. C., Labiche, Y., & Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, Vol. 32, pp. 608–624.
2. Basili, V., Caldiera, G., & Rombach, H. (1994). Goal question metric approach. *Encyclopedia of Software Engineering*, pp. 528–532.
3. Frankl, P. G. & Weiss, S. N. (1993). An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, Vol. 19, No. 8, pp. 774–787.
4. Frankl, P. G., Weiss, S. N., & Hu, C. (1997). All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, Vol. 38, pp. 235–253.
5. Frankl, P. G. & Weyuker, E. J. (1988). An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pp. 1483–1498.
6. Harrold, M. J. & Rothermel, G. (1994). Performing data flow testing on classes. *SIGSOFT Softw. Eng. Notes*, Vol. 19, No. 5, pp. 154–163.
7. Harrold, M. J. & Soffa, M. L. (1989). Interprocedural data flow testing. *ACM SIGSOFT '89 third Symposium on Software Testing, Analysis, and Verification (TAV3)*, ACM, New York, NY, USA, pp. 158–167.
8. Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. *16th International Conference on Software Engineering (ICSE-16)*, pp. 191–200.
9. Kakarla, S., Momotaz, S., & Namin, A. (2011). An evaluation of mutation and data-flow testing: A meta-analysis. *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 366–375.
10. Li, N., Praphamontriping, U., & Offutt, J. (2009). An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. *International Conference on Software Testing, Verification and Validation Workshops (ICSTW'09)*, pp. 220–229.
11. Mathur, A. P. & Wong, W. E. (1994). An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, Vol. 4, pp. 9–31.
12. Moreno, A., Shull, F., Juristo, N., & Vegas, S. (2009). A look at 25 years of data. *IEEE Software*, Vol. 26, No. 1, pp. 15–17.
13. Offutt, A. J., Pan, J., Tewary, K., & Zhang, T. (1996). An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, Vol. 26, pp. 165–176.
14. Rapps, S. & Weyuker, E. J. (1982). Data flow analysis techniques for test data selection. *6th international conference on Software engineering (ICSE'82)*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 272–278.
15. Vallespir, D., Apa, C., De Leïz $\frac{1}{2}$ n, S., Robaina, R., & Herbert, J. (2009). Effectiveness of five verification techniques. *XXVIII International Conference of the Chilean Computer Society*.
16. Vallespir, D., Bogado, C., Moreno, S., & Herbert, J. (2010). Comparing verification techniques: All uses and statement coverage. *Ibero-American Symposium on Software Engineering and Knowledge Engineering*, pp. 85–95.
17. Vallespir, D. & Herbert, J. (2009). Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. *Mexican International Conference on Computer Science (ENC)*, pp. 264–271.

18. **Weyuker, E. (1990).** The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, Vol. 16, pp. 121–128.
19. **Weyuker, E. J. (1984).** The complexity of data flow criteria for test data selection. *Information Processing Letters*, Vol. 19, No. 2, pp. 103–109.

Diego Vallespir is an Assistant Professor at the Engineering School of the Universidad de la República (UdelaR), Director of the Informatics Professional Postgraduate Center at UdelaR, Director of the Software Engineering Research Group (GrIS) at UdelaR, and a Researcher at PEDECIBA-Informatics. He holds an Engineering degree in Computer Science, a Master of Science in Computer Science, and a Doctor of Philosophy in Computer Science, all of them obtained from UdelaR. He has published several articles in international conference proceedings. His main research topics are empirical software engineering, software process, and software testing.

Silvana Moreno is a Teaching and Research Assistant at the Engineering School of the Universidad de la República (UdelaR). She is a member of the Software Engineering Research Group (GrIS) at the Instituto de Computación (INCO). Moreno holds an Engineering degree in Computer Science from UdelaR and a Master of Science in Computer Science from the same university. She is currently pursuing her doctoral degree in Computer Science.

Carmen Bogado is a Research Assistant at the Engineering School of the Universidad de la República (UdelaR). She is a member of the Software Engineering Research Group (GrIS) at the Instituto de Computación (INCO). She holds an Engineering degree in Computer Science from UdelaR.

Juliana Herbert is the founder and Director of the Herbert Consulting. PhD and Master in Computer Science, on Software Validation and Verification, by Universidade Federal do Rio Grande do Sul (UFRGS). Juliana led, as SCAMPI Lead Appraiser, 14 official CMMI-DEV appraisals by SEI/CMU (Software Engineering Institute of Carnegie Mellon University). She is Certified Scrum Master (CSM) and Certified Scrum Professional (CSP), by the Scrum Alliance. Juliana has been certificated as a PMI Agile Certified Practitioner (PMI-ACP), as a Project Management Professional (PMP) and a PMI Scheduling Professional (PMI-SP) by Project Management Institute (PMI). She is a 6 Sigma Black Belt by ASQ (American Society of Quality). Juliana is also a Senior Consultant, Appraiser and Instructor of models MPS-SW (Brazilian Quality Model for Software) and MPS-SV (Brazilian Quality Model for Services), accredited by Softex. She is a lead appraiser of CERTICS model (a product quality model, of the Brazilian government), accredited by the Federal Government of Brazil. She is the Vice-coordinator of the Technical Team Model (ETM) of Internationalization aspects of MPS models. Juliana works as a Senior Consultant in process management, software validation and verification since 1999. Juliana is an Associate Professor at the Universidad de la República del Uruguay, where she advises Master and Doctor students.

*Article received on 06/06/2014; accepted on 05/06/2015.
Corresponding author is Diego Vallespir.*