The Complementary Aspect of Automatically and Manually Generated Test Case Sets

Auri M. R. Vincenzi
Department of Computing
Federal University of São Carlos
São Carlos, SP, Brazil
auri@dc.ufscar.br

Tiago Bachiega
Department of Computing
Federal University of São Carlos
São Carlos, SP, Brazil
628247@comp.ufscar.br

Daniel G. de Oliveira
Instituto de Informática
Universidade Federal de Goiás
Goiânia, GO, Brazil
dangogyn@gmail.com

Simone R. S. de Souza
Institute of Mathematics and Computer Sciences
University of São Paulo
São Carlos, SP, Brazil
srocio@icmc.usp.br

José C. Maldonado
Institute of Mathematics and Computer Sciences
University of São Paulo
São Carlos, SP, Brazil
jcmaldon@icmc.usp.br

ABSTRACT

The test is a mandatory activity for software quality assurance. The knowledge about the software under testing is necessary to generate high-quality test cases, but to execute more than 80% of its source code is not an easy task, and demands an in-depth knowledge of the business rules it implements. In this article, we investigate the adequacy, effectiveness, and cost of manually generated test sets versus automatically generated test sets for Java programs. We observed that, in general, manual test sets determine higher statement coverage and mutation score than automatically generated test sets. But one interesting aspect recognized is that the automatically generated test sets are complementary to the manual test set. When we combined manual with automated test sets, the resultant test sets overcame in more that 10%, on average, statement coverage and mutation score when compared to the rates of manual test set, keeping a reasonable cost. Therefore, we advocate that we should concentrate the use of manually generated test sets on testing essential and critical parts of the software.

CCS Concepts

•**General and reference** → **Empirical studies;** *Validation; Verification;* •**Software and its engineering** → **Software testing and debugging; Empirical software validation;**

Keywords

Software Testing; Manual Testing; Automated Testing; Automated Test Data Generation

A-TEST'16, November 18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4401-2/16/11...$15.00
http://dx.doi.org/10.1145/2994291.2994295

1. INTRODUCTION

The test is a mandatory activity to ensure software quality. We know that earlier a failure is detected lower the cost of correction the faults responsible for the failure [5]. Usually, the test starts at the unit level, followed by the test at integration level and system level. The emphasis of this study is testing at unit level.

One important part of the testing activity is the selection of test cases to evaluate each unit under testing. Such test cases can be manually generated by the tester or using automatic test generators. In the latter, in general, it is created only the test data, i.e., the input of the test case because the generator has no knowledge about the product business rule and can not automatically generate the expected output for each input.

This work presents the results of an evaluation of adequacy, effectiveness, and cost of manually and automatically generated test sets for a set of 32 software products which implement data structures. We used the manually generated test cases from a previous work [8]. According to Souza et al. [8], a master's student created the manual test set for each program based on Equivalence Partitioning and Boundary Value Analysis testing criteria.

Additionally, we used three different test data generators – EvoSuite [11], Randoop [17], and CodePro [12] – to generate test sets automatically. We use EvoSuite [11] because it represents the state of the art of test data generator [11]. Randoop [17] because it is a random test data generator. Finally, we use CodePro [12] because, besides it is not an active project anymore, it integrates easily with Eclipse. Moreover, these tools generate test sets in JUnit format, making the data collection automation easier.

The primary purpose is to investigate the quality of the manual and automatically generated test sets isolated and combined concerning:

- adequacy, based on the statement coverage criterion;
- effectiveness, based on mutation testing; and
- cost, based on the number of generated test cases.

Observe that there are several ways to measure the cost, such as the time spend to generate the test set or the time to learn/set up automated test generators. Since the programs we are testing are relatively small, we decided to measure the cost in terms of number of generated test cases and explore these different cost metrics on further experiments.

We organized the paper as follows. In Section 2, we present the background necessary to understand this article. In Section 3 we discuss related work. In Section 4, we use the Goal Question Metric template to define the experiment we performed. In Section 5, we present the preparation for the experiment execution. In Section 6, we describe the experiment operational aspects. In Section 7, we introduce the data and analyze the results obtained. Finally, in Section 8, we draw the conclusion and point out future work.

2. BACKGROUND

A test case is a tuple $\langle I, E_O \rangle$ where I is input, also called test data, and E_O is the expected output concerning the unit specification. When we execute the unit with I it produces the resulting output, R_O. When R_O is equal to E_O, the unit under testing behaves as specified, on the other hand, a failure is detected by the test case.

Observe that the identification of the E_O is not an easy task and, in general, demands human intervention, making the cost of creating test sets for a huge number of units prohibitive, because the tester will spend a lot of time on selecting inputs and determine the expected output for each one of them.

Automated test generators generate input data I, but not the expected output data E_O. In general, E_O is defined later by the tester, who knows the unit specification and can infer E_O for each automatically generated I.

In the case of the automated generators used in this study, they adopted the following strategy. First, they read the source or bytecode of a given unit under testing (UUT). Second, they generate a set of input data SI for UUT. Third, they run UUT with each $I \in SI$ and collect the resulting output R_O. Forth, they assume R_O is the expected output E_O and create the test case $\langle I, R_O \rangle$.

Observe that, in this situation, all generated test case will pass in the current implementation of UUT, i.e., no test case will fail. These test cases, which assume the expected output as the current resulting output, is called by the authors of such tools as "regression test cases" [17, 12].

Automatic test data generation is useful when it can generate a few test cases which improve the coverage of a given testing criterion. Moreover, the main advantages of using automatic test data generators are that they can make the testing process faster, cheaper and reproducible because they can generate the same test set as many times we need very quickly. The primary disadvantage is that, in general, they are not able to generate test data to cover specific business rule, demanding human intervention in this case.

According to our results, manual and automatically generated test sets are complementary and should be used together to improve the test set quality.

3. RELATED WORK

Other researchers have already evaluated the relationship of manual and automatically generated test sets in a different context, using a different set of tools and programs. Below, we described some of these works.

Leitner et al. [14] agree that manual and automated tests are complementary. They developed a tool, called AutoTest, which tries to combine the best practices of both worlds. AutoTest assumes the developers adopt the concept of design by contract during software development. Contracts are executable preconditions, postconditions, and invariants embedded in the software source code. AutoTest uses the contract information to automatically generate test cases. Moreover, it also allows the tester to specify manually test cases for specific purposes such that testing complex business rules, while automatically generated test cases test other parts of the software. Although interesting, to get all benefits provided by the tool, developers must include the contracts in the software source code.

Smeets and Simons [20] evaluated the quality of manual JUnit test set and test sets generated by two automatic test data generator – Randoop and JWalk [19] – against mutants generated by MuJava [15]. They argued that automatic test generators, in general, are better than humans in the sense they ensure completeness in areas where tests are tedious to write, or when you are looking for particular cases, hard to discover. In their study, mutation score determined by both, manually and automated test sets, are low, below 70%, and all test sets can be considered incomplete on reaching high statement coverage and killing mutants.

Kracht, Petrovic, and Walcott-Justice [13] performed an empirical evaluation of automatically versus manually generated test suites, aiming at aiding developers in reducing the cost of the testing process. They used ten programs available from a software repository [10]. The programs were selected based on their size and availability of manually generated JUnit test sets. EvoSuite and CodePro were the automatic test generators they compared. The quality of test sets was evaluated based on branch coverage and mutation testing criteria. Their results showed that EvoSuite is better than CodePro regarding branch coverage and mutation score but manually generated test sets obtained a higher strong correlation between branch coverage [18] and mutation score. Although they used a different mutation testing tool for supporting mutation testing, the results obtained are similar to the ones we reported below. Moreover, they selected the set of program for experimentation based on their size regarding lines of code. Their large program with 18K lines of code has an average Cyclomatic Complexity (CC) of 2.82, and the smallest program with 783 lines has an average CC of 2.05. In our study, although we are using small programs regarding source code lines, the average CC is 3.1.

4. EXPERIMENT DEFINITION

We used Goal Question Metric template [4] for experimental study definition. Its main points are summarized as follow:

- Object of Study: The objects of study are manual and automatically generated test sets.
- Purpose: The purpose is to evaluate the complementary aspects of manual and automatically generated test sets.
- Quality Focus: The quality focus is the adequacy, effectiveness, and cost of test sets, evaluated against state-

ment coverage criterion, mutation testing criterion [1, 2], and number of test cases in the test set.

- Perspective: The perspective of the experimental study is from the researcher's point of view.

- Context: The researcher defined the experiment, considering a set of programs in the data structure domain and also performed the study. The study involves a participant (the first author) working on a set of objects, i.e., it is a multi-object variation study.

5. EXPERIMENT PLANNING

In the experiment planning, we set out the hypotheses and variables of the study. The plan is created and guides the conduction and analysis of the generated data. Next, we provide an overview of the main steps of the planning activity.

5.1 Context Selection

As mentioned in the introduction, the primary goal of our experiment is to investigate the complementary aspects of manually and automatically generated test sets. In the investigation, we measure the statement coverage of each test set, as well the mutation score, i.e., the number of mutants killed by the test set over the total number of generated non-equivalent mutants. Also, we evaluate the size of each test set.

We take a set of 32 Java programs which implement data structures, and the corresponding test sets from [8]. The manual test sets were generated based on the program specification, considering the functional criteria: class partition and boundary value analysis [18].

Thus, we classified our experiment as an offline study, performed by a graduate student, addressing a real problem – identification and comparison of adequacy, effectiveness, and cost of manual and automatically generated test sets in a particular context.

5.2 Formulation of Hypotheses

Our experimental study aims at evaluating the complementary aspect of test sets manually and automatically generated, looking for evidence that may define new hypotheses for future research.

We measured the Adequacy of a test set regarding statement coverage test criterion, the Effectiveness of a test set regarding the mutation score it determines, and the Cost considering the number of test cases in the test set. All these measures can be influenced by (i) the testing tools adopted; (ii) the skill(s) of tester(s) to adequately test the programs, and (iii) the size and complexity of the programs under testing.

Item (ii) is the same for all 32 programs because only one tester produced the manual test sets for these programs [8]. In this way, we defined the formal hypothesis under investigation as presented in Table 1:

Table 1: Hypothesis formalized – Adequacy, Effectiveness and Cost

Null Hypothesis	Alternative Hypothesis
There is no difference of adequacy among the manual (M) and automated test sets (A). $H1_0 : Adequacy(M) = Adequacy(A)$	There is a difference of adequacy among the manual (M) and automated test sets (A). $H1_1 : Adequacy(M) \neq Adequacy(A)$
There is no difference of effectiveness among the manual (M) and automated test sets (A). $H2_0 : Effectiveness(M) = Effectiveness(A)$	There is a difference of effectiveness among the manual (M) and automated test sets (A). $H2_1 : Effectiveness(M) \neq Effectiveness(A)$
There is no difference of cost among the manual (M) and automated test sets (A). $H3_0 : Cost(M) = Cost(A)$	There is a difference of cost among the manual (M) and automated test sets (A). $H3_1 : Cost(M) \neq Cost(A)$

5.3 Selection of Variables

Based on the context and the established hypotheses, we defined the independent and dependent variables for the experiment.

5.3.1 *Independent Variables*

An independent variable is any variable that can be manipulated or controlled in the process of experimentation [21]. The primary independent variables are: i) the testing tools adopted; iii) the testing criteria used; iv) the size and complexity of the programs under test; v) the tester's skills for applying the testing criteria; vi) the test sets. The last one, test sets, is the only factor of interest to the experiment. This factor has two treatments: manual and automated. The other variables in the experiment were set not to interfere with the results obtained.

5.3.2 *Dependent Variables*

The dependent variables are those in which it is possible to observe the effects of manipulation of the independent variables [21]. In our study, the dependent variable defined to describe the adequacy of the test sets is the statement coverage/program.

The effectiveness of the test set is the mutation score/program. We use mutation testing as a fault model to evaluate the capacity of the test sets on detecting a well-known set of faults represented by the mutants [1]. Greater the number of mutants a test set kills greater its effectiveness in detecting faults.

The dependent variables defined to describe the costs of the test set is the number of test cases/test set.

We also collected other metrics as detailed in Section 6.2.

5.4 Experiment Design

The experiment design has a direct impact on how to analyze the results. A suitable design also serves to minimize the influence of adverse factors on the results. As established earlier, our study addresses only one factor of interest, i.e., the test set quality.

Figure 1 summarizes the test sets evaluation strategy we follow. From a program P and specification S_P, we create a manual test set. Using the automatic test data generators, EvoSuite (E), CodePro (C) and Randoop (R), we generated three automated test sets for each program P.

Concerning Randoop, we fixed the maximum number of test case for each program as the maximum number of test cases generated by manual or generated using optimizations techniques. We also repeated the random test set generation 30 times and computed the average of the number of test cases generated and also the average of the statement coverage and the mutation score obtained.

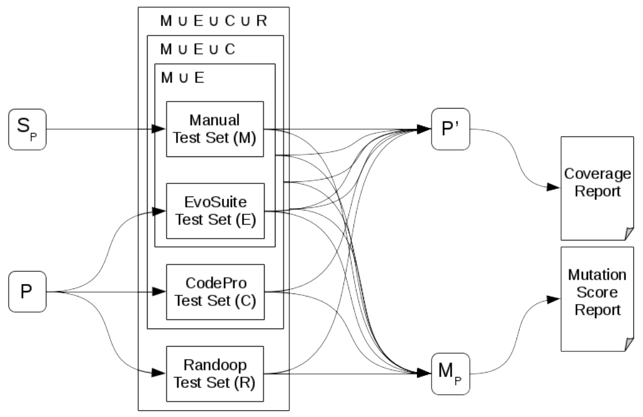

Figure 1: Test sets evaluation strategy.

Mutation testing is supported by Pitest (PIT) [7] which makes it possible to compute the mutation score and also statement coverage after each test set execution.

We combined the test sets to evaluate the complementary aspect between them. For instance, $M \cup E$, i.e., a test set composed of the union of manual with EvoSuite test set. In another test set, we combined $M \cup E \cup C$, representing the union of manual, EvoSuite, and CodePro test sets. Finally, we combined $M \cup E \cup C \cup R$, representing the union of manual, EvoSuite, CodePro, and Randoop test sets. In case of the random test sets, since we generate 30 test sets for each program, we get one of these 30 test sets at random to create the combined test sets. For all these combined test sets we also measured the statement coverage and mutation score.

6. EXPERIMENT OPERATION

The experiment operation includes the preparation of artifacts and tools, the execution of activities defined in the experiment plan and the validation of the data collected during the execution.

6.1 Preparation

As we mentioned, we used 32 Java programs to conduct our experiment. They implement traditional data structures [22]. Souza et al. [8] also used the same set of programs in their previous experiment, from where we also took the manual test sets for each program. More information about them can be found elsewhere in [8].

Before we use the programs in our experiment, we need to perform some adjustments. Since all the tools we used can be called by Maven [3] scripts, we create a Maven Project for each program using Eclipse [9]. Moreover, we also standardized the test set names such that, all test sets of a given tool have the same name, independently of the program under testing. The experiment is executed in a notebook Dell Inspiron, running Linux Ubuntu 16.04 LTS 64 bits, with 8 Gb of RAM and 1 Tb of hard disk.

After these adjustments and configurations, we wrote Python scripts to call Maven for program and test sets compilation, test sets execution, statement coverage and mutation testing computation and test report generation. Table 2 summarizes the tools and versions we used and their purpose in our experiment.

6.2 Execution

The first step in program execution is the computation of static metrics on each program (Table 3). For each program we compute the following metrics:

- the Non Commenting Source Statements (NCSS);
- the average Cyclomatic Complecity Number (CCN);
- the number of test cases on each test set Manual (M), EvoSuite (E), CodePro (C);
- the maximum number of test cases from M, E or C which is used to fixed the maximum number of random generated test cases;
- the average number of test cases generated by Randoop (R);
- the number of requirements demanded to cover statement coverage; and
- the number of generated mutants considering all mutation operators available in PIT.

Table 2: Tools version and purpose

Tool	Version	Purpose
JavaNCSS	32.53	Static Metric Computation
EvoSuite	1.0.3	Test Generator
CodePro	7.1.0	Test Generator
Randoop	3.0.1	Test Generator
Pitest	1.1.10	Mutation and Coverage Testing
Eclipse	4.5	Integrate Development Environment
Maven	3.3.3	Application Builder
JUnit	4.12	Framework for Unit Testing
Python	2.7.11	Script language

We did not spend time on generating manual test sets. In the work of Souza et al. [8], a master student created the manual test sets based on functional testing criteria Equivalence Partitioning and Boundary Value Analysis for all 32 programs. We took these test sets to use in our experiment [8].

The execution time for this experiment corresponds to the time to execute each test generator on each program, plus the time to run the scripts to collect testing reports about the statement coverage and mutation score for each test set individually, and test set combinations, as explained in Section 5.4.

An important information here is that we run the test generators using their default configuration, except for Randoop. For Randoop we generated 30 different test sets for each program to avoid any bias, fixing the maximum number of random test cases as the maximum number of test cases generated by the other ways. So, Randoop takes more time to create test sets and collect the testing reports.

6.3 Data Validation

Before we run the scripts for automatic data collection, we run all the tools manually in two different programs to make sure we will get the necessary information. Later we confront the manually collected information with the one gathered by the scripts, as we got no difference in the data we increased our confidence in our scripts and started the data collection for all 32 programs.

7. DATA ANALYSIS

First, we would like to characterize the programs we used in our experiment. From Table 3 we can see that they are, in general, simple programs, implemented by 1 to 3 classes (1.5 on average), 6.2 methods on average, summing up around 40,5 lines of code. Column CCM corresponds to the maximum cyclomatic complexity found in the methods of a given program, and CCA is the average cyclomatic complexity. CCM varies from 2 (minimum) to 9 (maximum) cyclomatic complexity, 4.8 on average. CCA ranges from 1.25 (minimum) to 9 (maximum) average cyclomatic complexity, 3.1 on average.

The last two column presents the number of test requirements demanded by statement coverage criterion and mutation testing criterion. In the case of mutation testing, we used all mutation operators implemented on PIT.

The columns labeled M, E, and C correspond to the number of test cases generated manually (M), automatically by EvoSuite (E), and automatically by CodePro (C). Column "R Limit" is the maximum value between M, E and C and is used as a threshold to limit the number of random test cases Randoop is allowed to generate in our study. Column R corresponds to the average number of test cases on 30 different random test sets Randoop generates for each program.

Observe that Manual test set has, on average, 9.8 test cases. EvoSuite generates 5.5 test cases on average; CodePro generates 13.5 test cases on average, and Randoop generates 10.3 test cases on average. Particularly for EvoSuite and CodePro, we consider these numbers of test cases manageable in the sense that if the tester wants to check if they are correct, it is a feasible task. Moreover, the standard deviation from M and E test sets is smaller than the ones presented by C and R. Therefore, in the case of cost, we should accept the alternative hypothesis $H2_1$ that there is a difference of cost between manual and automated test sets.

Table 4 presents the data about the statement coverage determined by each test set and the mutation score obtained.

From Table 4, considering individual test sets, from better to worst, we have M, E, R and C. On average, M obtained a coverage of 87.8% with standard deviation (SD) of 16.9%. E obtained a coverage of 81.7% and SD of 29.1. R obtained a coverage of 77.5% and SD of 17.3; and C obtained a coverage of 73.0 and SD of 22.6%. But there are specific situations where one test set performs better than other. We highlight in grayscale the cells where each test set performs better than the others. When the coverage is the same, we highlight all test sets cells with the same value.

Observing the pattern, E generates more test sets with 100% of coverage followed by M. Only for programs 27, 28 and 31, C obtained better results. They are among the most complex programs, and both M and E test sets could not overcome C. EvoSuite fails to generate test cases for program 24. Although Table 3 indicates that EvoSuite produced one test case for program 24, it is an empty test case as presented in Figure 2.

We could not identify the reason for this until the time of written this paper. Further, we intend to investigate this fact.

Considering the hypotheses we have established, we apply the Shapiro-Wilk normality test based on the data about coverage, mutation score (Table 4) and number of test cases (Table 3). The results indicate that the data does not have a normal distribution with a confidence level of 95% (p-value ≤ 0.05). This suggest the use of a non-parametric test and we used the Wilcoxon rank sum test to verify the difference among groups (the manual and automated test sets), considering a level of confidence of 95% ($\alpha = 0.05$). Column "p-value" shows the results of the test for each pair of test sets (Tables 5 to 7). Since we are testing multiple hypotheses, we applied the Holm-Bonferroni correction method which resulted in the column "corrected p-value" (Tables 5 to 7).

```
public class Evo {
  @Test
  public void notGeneratedAnyTest() {
    // EvoSuite did not generate any tests
  }
```

Figure 2: Empty test case generated by EvoSuite.

7.1 Analysis of Adequacy

Table 5 presents the Wilcoxon test on statement coverage criterion of manual and automated test sets for all 32 programs. The statistics suggest that concerning Manual (M) and EvoSuite (E) test sets, and Manual (M) and Randoop (R) test sets there is no statistical difference in terms of adequacy because the corrected p-value is above 0.05. Therefore, considering these test sets we have to accept the null hypothesis $H1_0$. On the other hand, there is a statistical difference between Manual (M) and CodePro (C) test sets because the corrected p-value is below 0.05 and, therefore, we rejected the null hypothesis, in this case, accepting the alternative hypothesis $H1_1$.

7.2 Analysis of Effectiveness

Table 6 presents the Wilcoxon test on mutation score of manual and automated test sets for all 32 programs. In this case, observe that the pairwise comparison of manual with the automated test, the test suggest there is a significant statistical difference, and we rejected the null hypothesis, accepting the alternative hypothesis $H2_1$ that manual test and automated test sets are different regarding effectiveness.

7.3 Analysis of Cost

Finally, Table 7 presents the Wilcoxon test on number of test cases generated manually or automatically for all 32 programs. In this case, observe that for M and E there is significant statistical difference in the number of test cases and we reject the null hypothesis $H3_0$, accepting $H3_1$. On the other hand, comparing M and C, and M and R there is no difference in the number of test sets of manual and automated test sets, and, therefore, we accept the null hypothesis $H3_0$.

7.4 Complementary Aspect of Test Sets

Based on the analysis above and in the grayscale patterns of individual test sets, we observed that it may have a complementary aspect complementary aspect between these test sets. In this sense, we decided to explore an incremental combination of test sets to check how much we could improve the adequacy and effectiveness. Observe that not all the test sets combinations were explored because we intend to investigate this in a further study.

For instance, considering the union of M and E, the coverage obtained is presented in column $M \cup E$ in Table 4.

Table 3: Static information of the Java programs

ID	Program	#Cls	#Met	NCSS	CCM	CCA	M	E	C	R Limit	R	#Req	#Mut
1	Max	1	1	8	3	3.0	3	6	3	6	5.3	4	14
2	MaxMin1	1	1	13	4	4.0	5	6	3	6	5.0	8	21
3	MaxMin2	1	1	14	4	4.0	5	5	4	5	4.2	8	21
4	MaxMin3	1	1	32	9	9.0	5	10	5	10	8.0	16	61
5	Sort1	1	1	11	4	4.0	3	4	4	4	3.5	10	21
6	FibRec	1	1	8	2	2.0	3	4	2	4	4.0	6	12
7	FibIte	1	1	8	2	2.0	3	4	2	4	4.0	6	12
8	MaxMinRec	1	1	26	5	5.0	7	3	4	7	6.7	13	41
9	Mergesort	1	2	22	6	4.0	6	1	2	6	4.0	16	56
10	MultMatrixCost	1	1	18	6	6.0	5	4	6	6	4.7	14	75
11	ListArray	1	4	20	3	1.8	14	7	7	14	6.4	12	29
12	ListAutoRef	2	4	23	2	1.3	7	3	6	7	3.3	12	21
13	StackArray	1	5	20	3	1.8	17	10	8	17	8.1	12	27
14	StackAutoRef	2	5	27	3	1.4	10	5	7	10	4.8	17	27
15	QueueArray	1	5	24	3	2.0	14	7	9	14	6.8	19	40
16	QueueAutoRef	2	5	32	3	1.6	10	5	8	10	4.8	23	32
17	Sort2	2	7	74	6	3.4	15	7	26	26	19.5	49	141
18	HeapSort	1	9	59	5	2.7	21	6	25	25	14.2	40	116
19	PartialSorting	1	10	62	5	2.5	25	14	26	26	15.6	42	120
20	BinarySearch	1	4	32	8	3.5	10	8	10	10	5.1	21	55
21	BinaryTree	2	11	85	7	3.0	14	10	5	14	6.7	48	145
22	Hashing1	2	10	61	5	2.1	11	7	13	13	6.1	35	88
23	Hashing2	2	12	88	7	3.2	11	10	26	26	11.9	51	162
24	GraphMatAdj	1	9	60	5	2.9	19	1	22	22	13.1	42	134
25	GraphListAdj1	3	16	66	4	1.6	18	5	22	22	12.7	34	95
26	GraphListAdj2	2	14	88	6	2.2	19	4	23	23	11.9	51	113
27	DepthFirstSearch	3	16	65	4	1.6	1	2	22	22	14.7	33	94
28	BreadthFirstSearch	3	16	65	4	1.6	5	3	22	22	14.7	33	94
29	Graph	3	16	65	4	1.6	4	3	22	22	14.7	33	94
30	PrimAlg	1	5	40	7	2.6	1	2	20	20	19.0	31	71
31	ExactMatch	1	4	55	8	6.3	16	4	53	53	53.0	40	205
32	AproximateMatch	1	1	24	7	7.0	8	5	14	14	14.0	19	88
Avg		1.5	6.2	40.5	4.8	3.1	9.8	5.5	13.5	15.3	10.3	24.9	72.7
SD		0.7	5.3	25.7	1.9	1.8	6.4	3.0	11.3	10.3	9.2	15.0	50.6

Table 4: Statement Coverage and Mutation Score per Test Set

ID	Statement Coverage per Test Set							Mutation Score per Test Set						
	M	E	C	R	$M \cup E$	$M \cup E \cup C$	$M \cup E \cup C \cup R$	M	E	C	R	$M \cup E$	$M \cup E \cup C$	$M \cup E \cup C \cup R$
1	75.0	100.0	50.0	93.3	100.0	100.0	100.0	85.7	71.4	28.57	38.6	92.9	92.9	92.9
2	87.5	100.0	87.5	94.2	100.0	100.0	100.0	81.0	66.7	52.38	25.7	85.7	85.7	85.7
3	87.5	100.0	87.5	91.7	100.0	100.0	100.0	81.0	28.6	52.38	24.9	85.7	85.7	85.7
4	93.8	100.0	75.0	84.8	100.0	100.0	100.0	72.1	26.2	13.11	11.2	75.4	77.0	77.0
5	90.0	100.0	50.0	60.7	100.0	100.0	100.0	81.0	81.0	42.86	17.6	81.0	90.5	90.5
6	83.3	100.0	83.3	98.9	100.0	100.0	100.0	100.0	100.0	83.33	95.8	100.0	100.0	100.0
7	83.3	100.0	83.3	98.9	100.0	100.0	100.0	100.0	75.0	83.33	95.8	100.0	100.0	100.0
8	92.3	100.0	23.1	88.5	100.0	100.0	100.0	80.5	19.5	7.32	18.2	80.5	80.5	80.5
9	93.8	93.8	12.5	66.9	93.8	93.8	100.0	91.1	21.4	5.36	17.9	91.1	91.1	91.1
10	92.9	92.9	92.9	90.5	100.0	100.0	100.0	38.7	30.7	48.00	27.8	50.7	52.0	52.0
11	100.0	100.0	75.0	71.7	100.0	100.0	100.0	65.5	69.0	24.14	36.1	72.4	72.4	72.4
12	100.0	100.0	91.7	95.0	100.0	100.0	100.0	71.4	66.7	52.38	58.3	71.4	71.4	71.4
13	100.0	100.0	100.0	81.7	100.0	100.0	100.0	74.1	74.1	44.44	61.1	77.8	77.8	77.8
14	100.0	100.0	94.1	88.4	100.0	100.0	100.0	66.7	77.8	62.96	63.0	81.5	81.5	92.6
15	78.9	100.0	73.7	68.4	100.0	100.0	100.0	57.5	85.0	22.50	39.6	85.0	85.0	90.0
16	100.0	100.0	78.3	89.0	100.0	100.0	100.0	68.8	65.6	34.38	64.0	68.8	68.8	78.1
17	97.9	70.8	64.6	64.0	97.9	97.9	100.0	85.8	35.5	32.62	21.0	87.9	91.5	91.5
18	97.5	80.0	85.0	56.9	97.5	100.0	100.0	76.7	39.7	43.97	19.9	77.6	79.3	81.9
19	28.6	97.6	28.6	57.4	97.6	97.6	97.6	29.2	45.8	6.67	22.4	61.7	64.2	70.8
20	100.0	100.0	71.4	52.5	100.0	100.0	100.0	69.1	67.3	23.64	15.0	80.0	83.6	83.6
21	81.3	58.3	37.5	46.5	81.3	81.3	81.3	55.9	21.4	8.97	14.3	56.6	56.6	57.2
22	100.0	100.0	77.1	85.0	100.0	100.0	100.0	59.1	58.0	28.41	45.7	64.8	64.8	63.6
23	80.8	94.2	82.7	92.4	98.1	98.1	98.1	47.5	51.2	30.86	47.5	67.9	69.8	67.3
24	100.0	0.0	83.3	83.6	100.0	100.0	100.0	73.1	0.0	52.24	52.9	73.1	73.1	73.9
25	97.1	64.7	82.4	79.5	100.0	100.0	100.0	68.4	37.9	45.26	43.5	70.5	70.5	73.7
26	96.1	78.4	68.6	60.5	98.0	100.0	100.0	73.5	30.1	35.40	30.9	75.2	81.4	81.4
27	60.6	30.3	81.8	74.3	69.7	87.9	100.0	30.9	10.6	30.85	33.4	37.2	40.4	60.6
28	39.4	18.2	81.8	74.3	39.4	81.8	93.9	13.8	11.7	30.85	33.4	14.9	34.0	54.3
29	84.8	18.2	81.8	74.3	84.8	100.0	100.0	38.3	10.6	30.85	33.4	39.4	41.5	61.7
30	93.8	43.8	50.0	28.1	96.9	100.0	100.0	38.0	5.6	5.63	1.4	38.0	38.0	38.0
31	97.5	72.5	100.0	96.1	100.0	100.0	100.0	40.0	21.0	51.22	41.5	42.9	57.1	62.0
32	94.7	100.0	100.0	92.3	100.0	100.0	100.0	29.5	31.8	37.50	33.0	35.2	38.6	40.9
Avg	87.8	81.7	73.0	77.5	95.5	98.1	99.1	63.9	44.9	36.0	37.0	69.5	71.8	75.0
SD	16.9	29.1	22.6	17.3	12.2	5.0	3.5	22.0	27.2	20.0	22.1	20.6	18.6	16.0

Table 5: Adequacy: statement coverage

Test Set Pair	p-value	corrected p-value
M-E	0.190849600	0.190849600
M-C	0.001129814	0.003389442
M-R	0.026056520	0.052113040

Table 6: Effectiveness: mutation score

Test Set Pair	p-value	corrected p-value
M-E	0.0005405501	0.0005405501
M-C	9.040424e-06	2.712127e-05
M-R	9.049378e-06	2.712127e-05

Table 7: Cost: number of test cases

Test Set Pair	p-value	corrected p-value
M-E	0.0002642351	0.0007927053
M-C	0.1756018000	0.3512037000
M-R	0.5370036000	0.5370036000

We can see that 11 out of 32 programs have coverages below 100%, and the average coverage reached is 95.5%. We highlight these programs to easy the evaluation of their coverage evolution. By combining an additional test set, column $M \cup E \cup C$, 7 out of 32 programs have coverage below 100% and the average coverage increase to 98.1%. Finally, by including one random test set to this combined test set, we obtained the results presented in column $M \cup E \cup C \cup R$ with only 4 out of 32 programs with coverage below 100% and the average coverage achieved is around 99%.

Regarding effectiveness, Table 4 presents the mutation score obtained by each test set individually and combined. In case of mutation score the values were not as higher as the coverage but, again M got better results on average with a mutation score of 63.9%, followed by E with a mutation score of 44.9%, followed by R with 37%, and C with 36%. The difference in the average mutation score is more significant than the difference in coverage, although the standard deviation for all test set is around 20%. One possible cause of this difference is that, in the case of mutation score, to distinguish the behavior of the original program and their mutants, PIT uses the assertions present on each test case. These scores are very low considering that usually it is easy to kill 80% of the mutants [6, 16].

When we write manual test cases, the tester has the knowledge about the program and their methods. He/She knows how to verify whether the correct result is computed and can write sophisticated assertion statements, making use of "getting" methods to evaluate the proper behavior of a `void` method, for instance.

On the other hand, automated test generator, in general, omit assert statement when generating test cases for void methods. Moreover, PIT measures the coverage and creates mutants from bytecode. In the case of CodePro, which generates test data from source code, when a class does not explicitly declare the default constructor, no test case is created to test it, but PIT works at the bytecode level. It will find the default construction in the bytecode, and will generate test requirements and mutants for it.

8. CONCLUSION

In this paper, we presented an evaluation of the complementary aspects of manual versus automated generated test sets over 32 data structure programs. We used three different automatic test data generators. In general, concerning the number of generated test cases, there are some differences due to the optimization strategy adopted by each generator. In the case of Randoop, we limited the number of generated test cases because, in 60 seconds, for the smallest program we used, it produces more than 30 thousand test cases.

Regarding adequacy, although on average, the manual test sets obtained better coverage, when analyzing the programs individually, EvoSuite test sets got 100% of coverage in 17 out of 32 programs, followed by manual testing. The statistical analysis suggested that there is no difference on adequacy between Manual, EvoSuite and Randoop test sets. Concerning effectiveness, the manual test sets obtained better mutation score in almost all programs. Only in 9 programs, the manual test set obtained scores below automated test sets, reaching a mutation score of 63.9% on average. The second best test set concerning effectiveness is EvoSuite test sets with a mutation score of 44.9% on average.

In summary, for this study, we observed that in relation to effectiveness, manual test sets is better on detecting faults modeled by PIT mutation operators. Effectiveness is the only hypothesis that statistical analysis was for the manual test with no exception. When looking for adequacy on statement coverage criterion, automated test sets generated by EvoSuite and Randoop obtained results as good as manual test set. The cost of manual and automated test sets is also closely related. Only for EvoSuite there was a statistical difference in the number of test sets when compared to manual test, with EvoSuite generating fewer test cases than human, keeping the same adequacy.

Finally, the statistical difference among the test sets regarding the effectiveness and data analysis suggested a complementary aspect between the manual and automatically generated test sets. An initial investigation by combining manual and automated test sets increased statement coverage from 87.8% on average, when using only manual test set, to 99.1%, when using the combined test set.

Effectiveness was also improved. We increased the mutation score from 63.9, only using manual test set, to 75%, when combining all test sets. Here, although there is more space for improvements, the complementary aspect of manual and automated generated test sets was also evident.

As future work, we intend to extend the experimentation to large programs, to investigate quality aspects about each automatic test generators and the reasons they failed on generating test cases for some programs/methods. Different combinations of manual and automated test sets must be investigated. We intend to establish an incremental testing strategy by combining manual and automatic test data generators aiming to reduce the number of faults previous to the software release. The idea is to keep a high statement/decision coverage concentrating manual test on critical parts of the application, complementing the coverage of other areas with automated generated test cases.

It is also important to implement minimization strategies for reducing the overlapping between manual and automated generated test data, contributing to speed up regression testing. As an alternative to avoid overlapping between manual and automated test cases, we intend to develop a selective instrumentation tool. This tool would receive as input the manual generated test sets, detect the uncovered statements after manual test set execution, and produce another version of the program, instrumenting only those not yet covered source code lines. When using the automatic test generators, we are interested in test cases which traverse uncovered parts of the source code to improve our test set.

9. ACKNOWLEDGMENTS

The authors would like to thank the Brazilian funding agencies FAPESP (Process N° 2014/15514-2), CAPES, and CNPq for their support for this research.

10. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *XXVII International Conference on Software Engineering – ICSE'05*, pages 402–411, New York, NY, USA, 2005. ACM Press.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, Aug. 2006.

[3] Apache Software Foundation. Apache Maven project. Página Web, June 2016. Disponível em: https://maven.apache.org/. Acesso em: 04/07/2016.

[4] V. R. Basili, G. Caldiera, and H. D. Rombach. *Encyclopedia of Software Engineering*, volume 2, chapter Goal Question Metric Paradigm, pages 528–532. John Wiley & Sons, Inc., 1994.

[5] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[6] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *7th ACM Symposium on Principles of Programming Languages*, pages 220–233, New York, NY, Jan. 1980.

[7] H. Coles. Pitest: real world mutation testing. Página Web, Jan. 2015. Disponível em: http://pitest.org/. Acesso em: 04/07/2016.

[8] S. R. S. de Souza, M. P. Prado, E. F. Barbosa, and J. C. Maldonado. An experimental study to evaluate the impact of the programming paradigm in the testing activity. *CLEI Electronic Journal*, 15(1):1–13, Apr. 2012. Paper 3.

[9] Eclipse Foundation. Eclipse ide. Página Web, June 2015. Disponível em: https://eclipse.org/mars/. Acesso em: 04/07/2016.

[10] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 178–188, Piscataway, NJ, USA, 2012. IEEE Press.

[11] G. Fraser and A. Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 33–36, New York, NY, USA, 2016. ACM.

[12] Google. Codepro analytix evaluation guide. WEB Page, 2010. Available at: https://google-web-toolkit.googlecode.com/files/CodePro-EvalGuide.pdf. Accessed on: 04/07/2016.

[13] J. S. Kracht, J. Z. Petrovic, and K. R. Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *2014 14th International Conference on Quality Software*, pages 256–265, Oct. 2014.

[14] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: The autotest experience. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261a–261a, Jan. 2007.

[15] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system: Research articles. *STVR – Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[16] J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi, and M. E. Delamaro. Evaluation N-selective mutation for C programs: Unit and integration testing. In *Mutation 2000 Symposium*, pages 22–33, San Jose, CA, Oct. 2000. Kluwer Academic Publishers.

[17] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[18] M. Roper. *Software Testing*. McGrall Hill, 1994.

[19] A. J. Simons. JWalk: A tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engg.*, 14(4):369–418, Dec. 2007.

[20] N. Smeets and A. J. H. Simons. Automated unit testing with Randoop, JWalk and MuJava versus manual JUnit testing. Research reports, Department of Computer Science, University of Sheffield/University of Antwerp, Sheffield, Antwerp, 2011.

[21] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Heidelberg, New York, NY, USA, 2012.

[22] N. Ziviani. *Project of Algorithms with Java and C++ Implementations*. Cengage Learning, 2011. (in Portuguese).