Chapter 5 - Evaluation

5.1 Subject characterization

5.2 Experimental setup

5.3 Results

5.4 Discussion

5.5 Threats to Validity

5.5.1 Construct validity

5.5.2 Internal validity

5.5.3 External validity

**How did you collect the Target subjects?**

We evaluated 38 subjects. All of them consisting of SPL evolution scenarios. 35 of them is randomly selected in the TaRGeT’s SVN repository.

We randomly chose 35 pairs of commits in TaRGeT’s SVN repository.

Write about Python Script and explain it!

As TaRGeT SPL was initially created as a single product, we had to find out the range of commits where we truly had a software product line. Look for the first commit where we do not only have source code, but mandatory artifacts of Software product line as Feature Model, Configuration Knowledge and Component Model.

We implemented a python script to walk through SVN commits and check which point the project had already SPL artifacts associated with the code.

After that, we randomly select pairs of commits to be evaluated.

Select one revision not used before

Svn first branch creation

Select one more revision 3 commits above to compound the evolution pair.

Add java project structure to both created branches

Run tool set for checking this evolution pair.

The python script selects one revision not used before and creates the first branch.

In the second step, the script selects one more revision 3 commits above to compound the evolution pair.

Automatically run our tool set for checking this evolution pair.

We definitely have an evolution pair,



**How did you collect the Mobile Media subjects?**

We also analyze 3 evolution pairs of Mobile Media [2], a product line for applications that manipulates music, video and photo on mobile devices.

These pairs have been already evaluated on Felipe's dissertation.

behavioral changes in transformations in Mobile Media were introduced by us.

We cannot guarantee that those pairs are representative enough for transformations applied in other product lines.

5.2 EXPERIMENTAL SETUP

We ran our experiment on an Intel Core i7-3820 processor at 3.6GHz with 32GB of RAM running a Linux distribution with the 3.2.0-23-generic kernel. The JVM is an OpenJDK, version 1.6.0 24, configured with a maximum heap of 20GB.

We use the command line version of the adapted Safe Refactor. It allows us to set a time limit

We defined a budget (e.g., the maximum total time allowed for

* testing by the user) of 5 min for both test generation tools
* The same time limit for both test generation tools



To generate TaRGeT products, we use the feature model and configuration knowledge available from its SVN history. We use the Mobile Media FM from its documentation, and we systematically translated its CK implementation from the build files.

5.3 RESULTS

This work has a contribution in two directions: we aim to reduce false-negatives and false-positives.

* Our tools detected all behavioral changes?
* Our techniques can really help developers early detect unsafe evolution scenarios?
* Experimental results Table (with standard deviation between x and y)
* Explain some evolution pairs.
* Show scenarios not detected by our tools

Our tool all together hits 32 out of 38 subjects. It means that, our tool set is effective in 84.21% of the experimental sample.

81.57 % of the experimental sample is behavioral changes. Our techniques could detect 80.64% of them.

These data show evidence that our techniques can really help developers early detect unsafe evolution scenarios.



Table x

Table x shows the summary of our experimental results. Each line corresponds to a subject.

The first 35 subjects are Target SPL. The last three subjects are Mobile Media.

The data are grouped by similar results.

Subject 1-6 are related to the false-negative of the *Impacted Class* approach.

We have a behavioral change in a particular method, however this negative change is not widespread to the backward impacted classes.

In subject 7 to 12, our tools could not expose the behavioral changes due to their limitation in stress structurally complex objects and perceive changes in output files.

Illustrate scenarios where it happens!

In the code below we have an essential method that need to be covered in order to identify the change, but our testing generation tools are incapable to pass an XML document in such format.

Subjects 13-21 show transformations where all tools detected the behavioral changes.

In subject 22-38, evosuite plays an outstanding role because it hits in all cases.

From subjects 22 to 26 and 31 to 35 only evosuite detects the Behavioral Changes.

22-26 Only evosuite detects the BC

31-35 Only evosuite detects the BC

Our tools quickly detected the compilation error introduced in some products in Subject 14, 15 and 16 by checking safe composition of the target product line.

On the subject 14, a method call passes four arguments to a method, which has been designed to receive only three parameters. This evolution makes the code uncompilable and accordingly our toolset quickly reports a non-refinement on the safe composition verification step.

On the subject 15, the evolution attempts to assign weaker access privileges to overriding methods. This code transformation is not well-formed because, we cannot override a method and reduce its visibility. The access modifier of an overriding method must provide at least as much access as the overridden method. Thereby, our tool easily detected a Non-Refinement.

On the subject 16, the evolution corrects a mistake that a class implements an abstract class instead of extend it. This change is correctly performed, however, we have a break in the code well-formedness, since our approaches verifies the safe composition of both, source and target SPL.

Safe composition guarantees that the product derivation process generates products

That still compile, named well-formed (wf) products, keeping the SPL consistent and detecting errors early [TBG11].



Create table to show the summary of our experimental results. And explain it!



Fazer tabela parecida!!

**18.42 %**

**10.52 %**

● EIC outperforms IC in regard to false-negatives due to dead code and masked-change situation as we explained before. Even though, EIC did not accuse any false-negative in this experimental sample, there are scenarios where it might happen. [Illustration on the next slide]



This figure shows a diagram of the Facade design pattern, which provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

In this scenario, we have an impacted class, which has a behavioral change in a method. Let’s suppose this code transformation is negatively spread to the facade class and our techniques are able to perceive this change even in such level. However, above the facade class, we have Graphical User Interface and even a Validation class associated with it. If the change performed on the subsystem class removes a contract that is already implemented on the validation on even inside the GUI, our *EIC* technique states a false-negative.

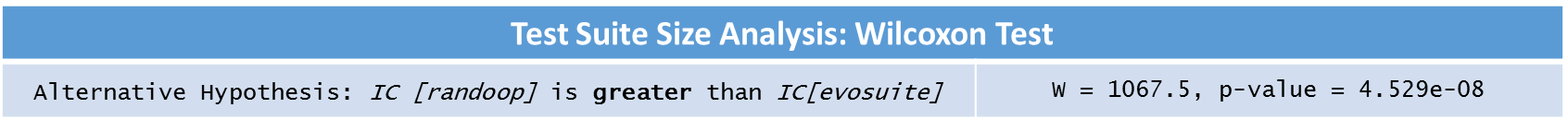
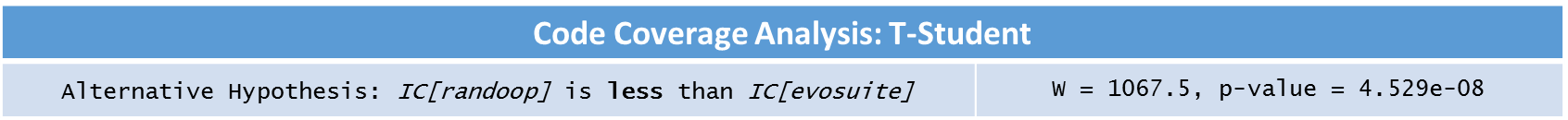
In this scenario we have a method behavioral change that is spread out to the facade however the SPL as whole is a refinement. Below we didactically show the code to illustrate it.

This is a hypothetical method for making a money withdrawal. The refactoring merges two closely coupled methods into one. This change ignores the withdraw limit requirement. If we run our approach to this evolution scenario, they accuse a non-refinement in the modified class and on the facade. However, the maximum withdraw contract is double implemented on the Validation class, which makes the whole spl a refinement because it do not negatively impact the spl products users.

● *Randoop* does not directly aim at code coverage. *Evosuite*, on the other hand, drives its search to obtain the highest achievable coverage and to keep the suite size as small as possible.

● As achieving greater coverage leads to higher probability of finding faults, *IC[evosuite]* produced more false-negatives than *IC[randoop].*

Our experiment outcomes show strong statistical evidence that the *IC[evosuite]* technique yields significantly better coverages and smaller test suites when compared to the previously implemented approach *IC[randoop]*.



**What statistical test you used and what you found:**

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ. It can be used as an alternative to the paired Student's t-test, when the population cannot be assumed to be normally distributed. [1]

The data for analysis were not distributed normally, therefore the non-parametric Wilcoxon signed rank statistical test was used to determine if there were any statistically significant differences between the test suite size of the techniques. A result is considered to be statistically significant if the p value is lower than 0.05. We test the hypothesis that *IC [ randoop ]* has a Test Suite Size greater than *IC [ evosuite ]*  and we found that really exists a meaningful difference between them since the p-value is smaller than 0.05

[1] Lowry, Richard. “Concepts and Applications of Inferential Statistics”.

**What statistical test you used and what you found:**

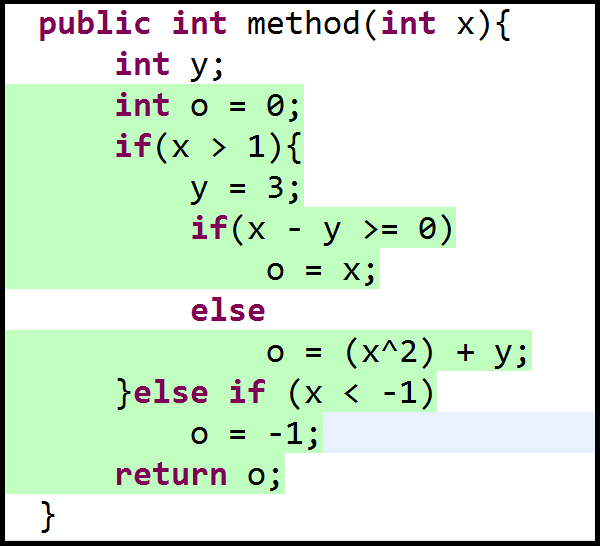
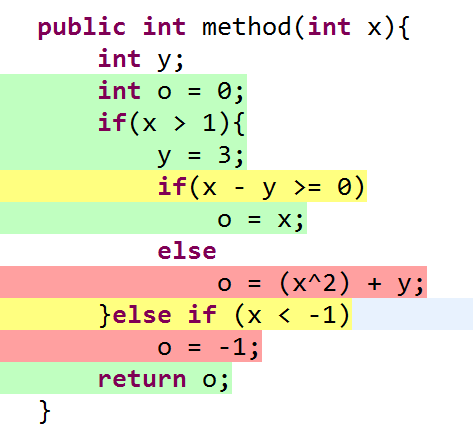
The T-Student test is a parametric test used in statistics in order to compare two or more independent samples aiming at verifying the existence of statistically signiﬁcant difference between the metric averages of these samples (Lehman, 1986).

We test the hypotheses that IC *[ randoop ]* code coverage is lower than *IC [ evosuite ]*, and the  t-Test confirmed our expectation that really exists a substantive difference between  the mean values of code coverage of these techniques.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

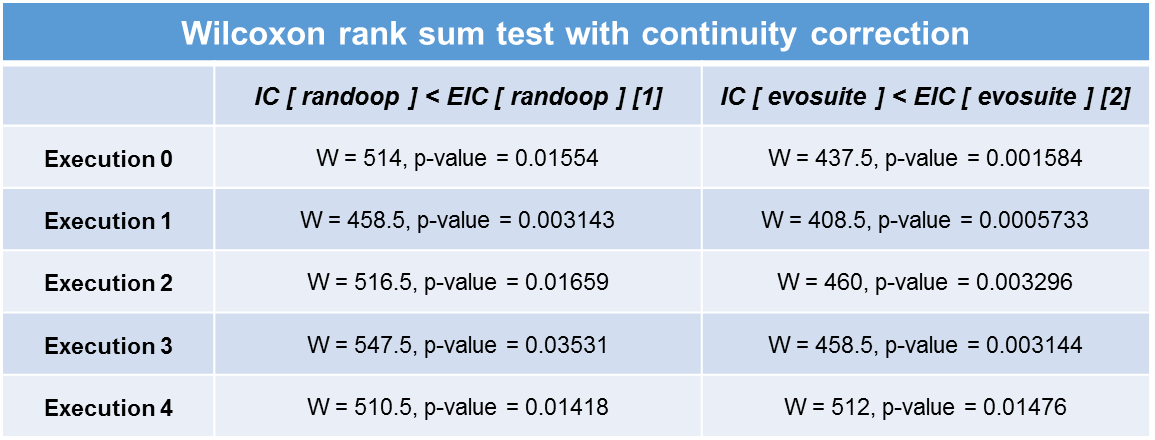
● If the P value is **lower** than **0.05**, you can reject the idea that the difference is due to chance, and conclude instead that the populations have different medians.

● If the P value **greater** than **0.05**, the data do not give you any reason to conclude that the overall medians differ. This is not the same as saying that the medians are the same. You just have no compelling evidence that they differ.



In our subject sample, *EIC* decreases false-negatives, however it **takes more time** to check the evolutions due to the additional time for calculating the backward impacted classes. We conclude that, because we set the same testing generation time for both techniques and also there is no significant difference between the amount of tests. In addition, Wilcoxon Test confirms this expectation in 5 out of 5 executions with 95% of confidence level.

|  |  |
| --- | --- |
| **Test Suite Size: Wilcoxon Test** | |
| Alternative Hypothesis: IC [ randoop ] is **greater** than EIC [ randoop ] | W = 643.5, p-value = 0.36 |
| Alternative Hypothesis: IC [ evosuite ] is **greater** than EIC [ evosuite ] | W = 585.5, p-value = 0.6271 |



95% of Confidence Level

**Alternative Hypothesis [1] :** *IC[randoop]* has a significant time lower than *EIC[randoop] .*

**Alternative Hypothesis [2] :** *IC[evosuite]* has a significant time lower than *EIC[evosuite].*

Which one reduce False-Positives ?

42.10%

15.78%

57.89%

44.73%

We reduced 62.5% of False-Positives using evosuite in Impacted Classes and 22.72% in Extended Impacted Classes.

**Why evosuite reduces more false-positives in *IC* than *EIC* ?**

Our experimental sample shows that is more difficult to perceive behavior changes on the backward impacted classes, because they usually have complex object associations and  specific file dependencies which satisfy particular syntactic and semantic properties. Moreover, neither evosuite nor randoop supports test generation for structurally complex inputs files. Therefore, *EIC* indicates a slight difference of false-positives between both tools,  because they have this common limitation and face the same challenges. Additionally, it might also occur in *IC*, but in smaller scale as the data confirms.

**Why evosuite reduces more false-positives in *IC* than *EIC* ?**



Moreover, evosuite is always better than randoop. It means, when the former fail, the latter do as well.

Evosuite uses an evolutionary technique in which, instead of evolving each test case individually, it evolves all the test cases in a test suite at the same time. Since coverage goals can be infeasible such that there exists no input that would cover them, this strategy prevents waste of time in unreachable branches. This evosuite approach outperforms randoop, because the latter does not even aim at code coverage. This fact makes evosuite better than randoop at code coverage.

**evosuite** average coverage is **38.78%** higher than **randoop**

*IC[ randoop ]* ***vs.*** *IC [evosuite ]*

*EIC [ randoop ]* ***vs.*** *EIC [ Evosuite ]*

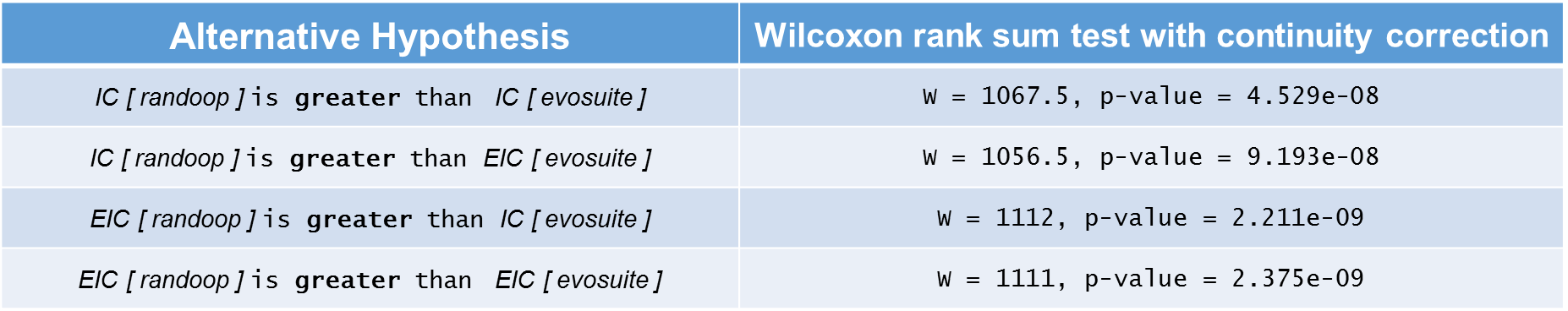
● **Second Reason**

*evosuite* includes the total length of a test suite as a secondary optimization goal. As the stopping conditions are based on coverage achievement, *evosuite* therefore minimizes test suites as a post-processing step. This feature enhances test readability and performance, because the test suite is smaller and accordingly the techniques spend less time to compile and run tests.

In our experimental sample, randoop generated test suite size 99.93% bigger than evosuite

|  |  |
| --- | --- |
| Testes per second in average | |
| randoop | 37 |
| evosuite | 0.02 |

● The Wilcoxon statistical test confirmed that exists difference extremely significant among all comparisons of randoop against evosuite regarding test suite size.



● The Wilcoxon test also showed that exists statistical significance among all comparisons of randoop against evosuite regarding the time spent to check product line evolutions.

● Techniques combined with **randoop** takes more time to check evolutions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Wilcoxon rank sum test with continuity correction** | | | | |
|  | ***IC [ randoop ] > IC [ Evosuite ]*** | ***IC [ randoop ] > EIC [ Evosuite ]*** | ***EIC [ randoop ] > IC [ Evosuite ]*** | ***EIC [ randoop ] > EIC [ Evosuite ]*** |
| **Execution 0** | W = 1229.5, p-value = 6.921e-08 | W = 1004, p-value = 0.001722 | W = 1269.5, p-value = 6.616e-09 | W = 1098, p-value = 4.776e-05 |
| **Execution 1** | W = 1212, p-value = 1.83e-07 | W = 979, p-value = 0.003848 | W = 1266.5, p-value = 7.944e-09 | W = 1090, p-value = 6.721e-05 |
| **Execution 2** | W = 1203.5, p-value = 2.908e-07 | W = 987.5, p-value = 0.002952 | W = 1246.5, p-value = 2.607e-08 | W = 1062.5, p-value = 0.0002059 |
| **Execution 3** | W = 1194, p-value = 4.825e-07 | W = 972, p-value = 0.004767 | W = 1239, p-value = 4.023e-08 | W = 1058.5, p-value = 0.0002407 |
| **Execution 4** | W = 1086, p-value = 7.955e-05 | W = 921.5, p-value = 0.01935 | W = 1152, p-value = 4.058e-06 | W = 1001, p-value = 0.001905 |

Discussion

● Our experimental results showed that *IC [evosuite]* is slightly more effective than all techniques, has the lowest number of false-positives and the shortest time to check the evolutions. On the other hand, it produces more false-negatives. *EIC [evosuite]* reduces false-negatives, however it yields more false-positives than *IC [evosuite]* .

● The outcomes show that no technique strongly stands out, each one is better in different scenarios. *IC* outperforms *EIC* when the latter has to expose failures in structurally complex objects and *EIC* surpass the former in dead codes and masked-change situations.

● For this reason, we are considering as a future work, design a hybrid solution to check backward impacted classes one level below of components which require convoluted objects. This strategy surely outputs better results.

*IC [evosuite]*

*IC [randoop]*

*EIC [evosuite]*

*EIC [randoop]*

1

3

2

4

Performance Podium