Paper Title

**A Comparative Evaluation on the Quality of Manual and**

**Automatic Test Case Generation Techniques for Scientific**

**Software - A Case Study of a Python Project for Material Science**

**Workflows**

* To investigate the efficacy of automated test case generation
* Project Under research: Atomic Simulation Environment
* Branch and mutation coverages
* None of the automated TCG algorithms come close to manual
* Evosuite – achieve good levels of coverage
* Pynguin – algorithms supported: DynaMOSA, MIO, Whole test suit generation for PYTHON
* RQ: ﻿How do the automatically generated test suites of the TCG algorithms compare against each other as well as against a manually created test suite?
* TCG is an optimization problem (with metrics: line, branch, coverage) employed using limited time and computation budget.
* Whole Test Suite Generation Approach - tries to reach a high coverage and minimize the size of the test suite.
* MOSA – an adaptation of the whole test süite approach with a change in how a test case is selected (the shorter test case is selected for the next generation)
* DynaMOSA – computes test cases for tärget that can be immediately reached. It reduces the number of targets that need to be covered at any one time.
* MIO – out the created test candidates, it removes the worst test case and replaces it with the next better one encountered during the search. It also selects the shorter of the two test cases.
* ﻿It was shown that MIO can, on average, cover more than Whole Test Suite Generation and in some cases is also better than MOSA
* FD-random another non-deterministic approach provided in pynguin is also tested.
* **Tools and versions used**: Python 3.8.11 , pytest 6.2.4 , pytest-cov 2.12.1 , pynguin 0.9.2 ,cosmic-ray 8.3.5
* We kept the tests small but numerous so that localizing root causes for any failure is easier
* Neither the baseline nor our manually created test suite shows any variance as the test suites themselves do not change.

Branch coverage:

o Best Value was reached by DynaMOSA however, overall MIO performed better on average than DynaMOSA

* All algorithms reach similar values in the very beginning. This is due to the fact that all methods need to employ a random set of test cases which they can then mutate.
* Close to the end of our 600s search budget, we note that indeed all four algorithms reach plateaus: Any further improvement seems to become increasingly difficult for the algorithms to find.
* While in FD-random this high level of variance is kept throughout the entire runtime, the other three algorithms show a slow decrease in variance over the runtime of 600s.
* These general experimental results are in line with data describing the behavior of the selected algorithms in the literature.
* Previous Works: Daka et al argue about the naming of variables in the generated test cases which is pretty evident in the research done in this paper.
* TCG algorithms are better than the baseline test süite but way behind the manual test süite. Meaningful naming conventions provide the developers with more info in the manual test suite.

Comparisons between TCG algorithms

MIO reaches significantly higher branch coverage than others:

o ﻿DynaMOSA (𝑝 = 0.007396)

o ﻿Whole Suite Generation (𝑝 = 1.019 · 10−6), and

o ﻿FD-random (𝑝 = 2.2 · 10−16).

While – as expected – all of the four investigated algorithms provide a significantly better test suite in terms of branch coverage than the original baseline (MIO: 𝑝 = 0.03833, DynaMOSA: 𝑝 = 0.04466, Whole Suite: 𝑝 = 0.04575, and FD-random: 𝑝 = 0.04736), the highest branch coverage reached by any run of any of the four algorithms was only about 61%

a reason for the low coverage and sensitivity of the test suites may be system states that are difficult to reach.

The statistical analysis shows that the automatically generated test suites are significantly worse than the manually created test suite by a large margin. We hypothesize that this may be due to the dynamically typed language, as well as the employed program that requires complex structured inputs for a certain method

Metaheuristic search technique

To mitigate the uncertainty stemming from the randomness in the algorithms, they were run 50 times

**Mutation testing**, also known as code mutation testing, is a form of white box testing in **which testers change specific components of an application's source code to ensure a software test suite will be able to detect the changes**. Changes introduced to the software are intended to cause errors in the program

NEED FURTHER WORK:

* To verify whether creating an ensemble test suite out of all automatically generated test cases would achieve higher coverage and mutation score than a manually-written one.
* TO clarify why MIO outperforms DynaMOSA in this particular use case
* More work is needed for TCG when dealing with programs that are written in dynamically typed languages such as Python and which require complex structured inputs.

Project Setup on Machine:

Pynguin: 0.25.2

Python: 3.10.0

PIP: 21.2.3

Commands:

//Creating virtual env is recommended

//Set this temporary env var for pynguin to work

export PYNGUIN\_DANGER\_AWARE=1

//To run pynguin and start testing a module

pynguin --module-name queue --project-path . --output-path . --seed 1629381673714481067 -v

**How and why of pyguin:**

It uses search-based test generation to generate tests that maximize code coverage.

We start with a module that has python code and we finish with test\_cases for the module generated with pynguin.

From starting with the inputs and eventually coming to the results penguin goes through numerous stages which allow modifications easily.

**STAGES:**

1. Analysis:

Analyses the module and gathers information about its functions, classes, and methods defined in the module.

1. Test Cluster:

These pieces of information are included in the so-called “Test Cluster” which serves as a repo for the test generation tool for functions or classes to be called during the generation of test cases.

**Note:** Since most of the modules on the test are not standalone, this test cluster also considers the modules being imported and the functions, classes, and methods being used in the ON-test Module to be used as inputs in the test case generation step for pynguin.

1. Test Case Generation

From the test cluster information, pynguin starts to generate test cases using most of the well-established TCG algorithms. (DynaMOSA, Whole suite).

Each generated test case is then applied in a loop on the module under test to check the coverage rate of statements and branches yielding the best coverage rate. Some of these test cases might crash the application. Inside the loop, the generated test cases are improved after each run, and always the target is to get the best branch and statement coverage.

This process stops once a configurable stopping condition is satisfied, such as a time limit or a predefined amount of algorithm iterations. It is also possible to stop the generation after all coverage goals have been met, which means the generated tests achieve 100 % coverage

1. Assertion Generation

However, the test cases are still only a sequence of statements. Now the process moves on to mutation generation which aims to generate regression assertions that assert the behavior of the module under test.

1. Mutation Generation

We test the module under test with mutants generated for those cases where the assertions change and discard mutants for which no change occurs.

1. Test Case Export

Export the test cases to the module under test.

**Similar Tools:**

TSTL , CrossHair2, Klara , Auger generate test cases too but very restrictive way.

1. Ho

**NEW Findings**

From the context, Pynguin extracts the types they define by searching for those class definitions that are available in the namespace of the module under test. These types are then used

as input-type candidates during the test-generation phase. Pynguin selects classes, methods, and functions from the test cluster during the generation to build the test cases.

For example: For a function triangle inside a module under test

1. Pynguin add a statement representing a method call to the function.
2. It aims to fulfill the requirements in a backward fashion. E.g.