Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Evaluating and Improving the Test Coverage of scientific Python Software

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Jonas Trappe
geboren am: 16.05.1997
geboren in: Berlin

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

eingereicht am: ..... verteidigt am: .....

Contents

1 Introduction

For modern software, testing is one of the most crucial parts of the development process. In addition to classic unit testing done manually, developers can resort to automated approaches. Powerful tools exist both as mature research prototypes, such as EvoSuite[1] or Randoop[2] for Java, or even as commercial applications, like TASMO[1] for C/C++. However, few of such tools exist for Python. To the best of our knowledge, the only real unit test generation tool for Python is Pynguin[3]. This is interesting, because Python is one of the most popular programming languages used today. As of writing, Python holds the first place on the IEEE spectrum rating[2].

This raises a number of questions:

- What problems does automated test case generation face in Python?
- Do performance expectations established in other languages hold for the same techniques in Python?
- How do established automated test case generation algorithms for other languages perform compared to each other in Python?

Due to its syntactic simplicity, Python is popular among scientists and people with no direct computer science background. Improvements towards automated testing could greatly help anyone using the language to increase their code's quality. Python's flexibility also comes with challenges for testing, which we would like to explore and discuss.

The authors of the aforementioned Pynguin name dynamic typing as a core issue for automated test case generation in Python. As types of variables are typically not known before runtime, a test case generator has more options to choose from. This issue is discussed more in-depth in section 2.3. A comparison between different techniques has only been done to a limited extent by Lukasczyk et al.[3].

Apart from test case generation, we want to examine the usefulness of other testing practices in Python, such as the usage of mock testing[4]. Mock testing means replacing an object or component of the system under test (SUT) with a less capable but also less resource intense object with predefined behavior. This is typically done to test the context of said component, but mocks can also be used to save runtime during testing.

Open challenges addressed by the thesis project In this thesis, we want to examine how testing in Python can be made more efficient. We therefore want to compare different automated test generation techniques in regard to the test coverage they achieve. We also want to explore the usefulness of mock objects in a Python pipeline in order to save time in the testing process.

[1] Piketec: Automatic test case genration in TPT, `https://piketec.com/tpt/testcase-generation/`

[2] IEEE Spectrum: Top Programming Languages 2021 `https://spectrum.ieee.org/top-programming-languages/`

- RQ1: How do different test generation techniques compare to each other on a Python program?
- RQ2: To what extent can mock objects be used in a Python testing pipeline?

To accomplish our goals, we evaluate testing techniques on the material science program elastic2. We have selected the tools Hypothesis[3] and Pynguin[3] for test case generation. To answer **RQ1**, we compare the selected automated test generation techniques to a gold standard test suite for elastic2. We evaluate if assumptions made for other programming languages hold in Python. For example, random test case generation usually performs worse than genetic algorithms. We also implement mock objects in the elastic2 test suite to skip expensive calculations during testing.

[3]HypothesisWorks: Hypothesis testing framework, `https://hypothesis.readthedocs.io/en/latest/`

2 Preliminaries

This section contains information about concepts mentioned in the thesis. It also show descriptions for the tools used in this project and reviews of other research done on the topic of testing Python software.

2.1 Test Coverage

Test coverage is a term that describes how much of a program's source code is tested by its test suite. Test coverage is usually given in percent or a value between 0 and 1. A low value can generally be interpreted as worse. There are different metrics to measure test coverage.

Definition 1 (line coverage)**.** One of the most simple metrics is line coverage[1], which is the percentage of lines that are executed by the test suite. It can take values between 0 and 1, with 1 meaning all lines of the program are covered. When $CoveredLines$ is the number of lines reached by a test suite and $TotalLines$ the number of executable lines in a program, then line coverage C_{line} is defined as:

$$C_{line} := \frac{CoveredLines}{TotalLines}$$

Line coverage is useful to determine the completeness of a test suite. A lower line coverage means a larger portion of the SUT is not reached by test cases at all.

Definition 2 (mutation coverage)**.** Another metric is mutation coverage[5], for which randomly modified variants of the program under test are created. These mutants then have to be "killed" by the test suite, meaning that for each mutant, at least one test case needs to be able to distinguish between the original program and the mutant.

There are different criteria for when a mutant is viewed as "killed". To "kill" a mutant under **weak mutation coverage**, the mutated program p' must differ in its state from the original program p when tested, e.g., by different values being assigned to variables. **Strong mutation coverage** additionally requires the the difference between p and p' to be visible in the tests output. Mutation coverage C_{mut} is generally defined as

$$C_{mut} := \frac{KilledMutants}{TotalMutants}$$

with $KilledMutants$ being the number of mutants detected by the test suite under the weak or strong definition and $TotalMutants$ being the total number of available mutants for p. Again, C_{mut} can take values between 0 and 1, with 1 meaning the program's test suite "killed" all mutants.

Mutation coverage can be used to determine the sensitivity of a test suite, meaning that a test suite with a high mutation coverage is more likely to detect errors. We use the Python package `mutmut`[4] in this thesis to perform mutation testing.

[4]mutmut - python mutation tester `https://mutmut.readthedocs.io/en/latest/`

Different metrics can produce different results. Consider the example taken from Medium[5] in Listing 1.

```
# returns False if temp is smaller than 1000
def isDangerous(temp):
        if(temp <= 1000): # bug: <= instead of <
                return False
        else:
                return True

def isDangerous_test():
        assert isDangerous(1500) == True
        assert isDangerous(200) == False
```

Listing 1: An example with a line coverage of 1 which contains an undiscovered error

In the test function, all lines of `isDangerous` are executed at least once, the line coverage is 1. However, the edge case of `temp = 1000` is not tested. This leads to the bug not being detected when the input is `temp = 1000`. Mutation testing might have replaced the $<=$ in `isDangerous` with $<$, creating a mutant that does change the functionality of the program. As this is not detected by the test suite, the mutation coverage would not be 1.

However, using only mutation coverage could lead to other issues. Generating every possible mutant gets increasingly expensive with program size. And even if a large number of mutants is available, the risk of "equivalent mutants" exists. They are syntactically different but semantically equal to the original program, meaning they cannot be "killed" by any test case. This highlights the importance of not relying on a single coverage metric.

2.2 Test Case Generation Algorithms

Coverage metrics are not only useful to assess how good a test suite is, but can also provide guidance for automated test case generation techniques. These techniques often use a genetic algorithm[6] to generate tests. Genetic algorithms are based on evolutionary principles found in nature but applied to computer science. Such algorithms "evolve" a population by applying mutations, random alterations performed to a single candidate, and crossovers, building a new candidate from two existing ones. How good a candidate is for the task it is supposed to fulfill is assessed by a so-called **fitness function**.

For test case generation, the candidates are either test cases or entire test suites. The fitness function calculates how close a candidate is to reaching given coverage goals.

[5]Medium: Tests Coverage is Dead — Long Live Mutation Testing `https://medium.com/appsflyer/tests-coverage-is-dead-long-live-mutation-testing-7fd61020330e`

Whole test suite generation [7] is the name of an algorithm pioneered by Fraser et al. for the tool EvoSuite[1]. Its genetic algorithm optimizes for multiple objectives at the same time, which was not done by others before. Their goal was to develop an algorithm that focuses on a given coverage metric, for example branch coverage, and at the same time provides a minimal test suite. The fitness function takes into account both of these objectives. For the context of this thesis, we refer to the whole test suite generation implementation used in Pynguin[3]. This implementation targets branch coverage and has a fitness function adapted to the way Python code is compiled. We will refer to the algorithm as *whole suite*.

DynaMOSA [8] (Dynamic Many Objective Sorting Algorithm) is a genetic algorithm that considers a coverage goal as a multi-objective problem. When optimizing for branch coverage for example, it uses one variable in the fitness function for each branch in the SUT. The search for new test cases is then only directed towards objectives that are not covered yet. Unlike in *whole suite*, the population which is evolved consists of test cases rather than test suites. Once an objective is achieved (e.g., a branch is reached), the corresponding test case is archived. This means that already found solutions are preserved and only discarded if better ones are found, for example a shorter test case with the same coverage. All these features are already present in MOSA, the predecessor to *DynaMOSA*. *DynaMOSA* adds dynamic target selection. This means that targets will only be taken into account if they can be reached by the current test population. In the example of two nested `if` statements, *DynaMOSA* will only try to cover the inner statement if the outer one has already been covered.

MIO [9] (Many Independent Objectives) is another multi-objective algorithm that uses evolutionary techniques. As in *(Dyna)MOSA*, it assigns a fitness value to each objective and uses an archive to store test cases for covered targets. *MIO* also keeps a population of test cases rather than test suites. The population size scales with the number of objectives. Additionally, *MIO* will start a "focus phase" after a certain amount of the search budget is consumed. This means *MIO* will focus on fewer objectives at a time later in the test case generation process. The implementation in Pynguin[3] which we use in this thesis will by default start the focus phase after 50% of the testing time has elapsed. From then on, *MIO* will apply more modifications to already discovered test cases before creating a new one. This is done to cover more objectives that are already close to being covered, rather then finding new objectives, which are less likely to be covered before the testing time elapses.

Feedback-Directed Random Testing does not rely on test coverage metrics for guidance or any sort of genetic selection. Originally created for the tool RANDOOP[2] in Java, such a strategy was also implemented in Pynguin, which we will refer to as *FD-random* in this thesis. It is relatively simple: test generation starts with two empty sets of test cases, one for passing and one for failing tests. Test cases are created by generating random statements using the SUT, or by randomly altering test cases that

passed previously.

Test generation algorithms in comparison Campos et al. conducted a study[10] on 117 Java programs, ranging from 14 to 16.624 statements, to compare different test case generation techniques using EvoSuite[1]. They found *DynaMOSA* to clearly outperform any other algorithm. *Whole suite* did perform better than *MIO* in their experiment, while all mentioned algorithms outperformed the random approach.

2.3 Test Case Generation in Python

Test case generation is an established part of software engineering, with prominent tools such as EvoSuite[1] for Java. However, test case generation techniques usually rely on type information, which is absent in Python. In order to generate tests, a test generator has to create objects or call functions from the system under test. This is done either randomly or guided by a fitness function to achieve a higher coverage under one or more metrics. As soon as an object constructor or function has parameters, the test generator has to decide what to pass to it. With type information, this is relatively easy: the generator can just take an object or a variable in the current scope that matches the parameter type and pass the object to the function. If such an object does not exist, the generator will create one by calling the constructor and creating necessary parameters for it if needed. Without type information, the generator can theoretically pass any object to a function. This includes any primitive type, as everything is an object in Python. That means a generator has to guess which objects are to be passed to a function, which drastically increases the decision space for the generator.

Lukasczyk et al.[3] have shown that including type information can positively affect the achieved coverage by test case generators. They tested their *FD-random* and *whole suite* algorithms on ten Python modules that had between 85 and 1715 lines of code using Pynguin 2.4 The *whole suite* approach performed better across the board. Its average line coverage was around 5% higher than that of the *FD-random* approach for any runtime up to 600 seconds. They also found that the availability of type information at compile time did generally increase the performance of both techniques. However, the degree of improvement was dependent on the system under test. Modules that required specific types had larger increases in test coverage when adding type information compared to those using mostly primitive types.

2.4 Tool: Pynguin

The framework Pynguin[3] was developed at the University Passau for automated unit test generation in Python. It is generally comparable to the Java tool EvoSuite[1]. Pynguin implements a range of test generation algorithms which have already successfully been used in other languages. These include the previously mentioned (section 2.2) *FD-random*, *whole suite*, *MIO*, and *(Dyna)MOSA*. Apart from the implementations of those algorithms, Pynguin offers a number of utility features. The test coverage, either as line or branch coverage on a bytecode level, can be evaluated during testing. Test

generation can be executed with or without regards for type information. Pynguin can also generate assertions for test cases.

Pynguin is built to take a single module, usually the contents of one Python file, and generate units tests for it. It will run either a given time or until a certain test coverage is achieved. The resulting test cases are saved, divided into successful and failed tests. A test is considered as failed if it raises an exception or breaks an assertion.

2.5 Tool: Hypothesis

Hypothesis[6] is a property-based testing framework for Python. It can automatically generate input values for test cases according to some specification. This addresses the problem of dynamic typing in Python, as the user decides which types and even which values an input variable can take. For example, a test function can be told to only use positive integer values in as inputs for testing. Hypothesis will then generate positive integers, trying to find examples that either lead to exceptions or break the assertion made in the test case.

This approach however still requires the developer to not only write test cases, but also decide which types its variables have. The automatic input generation is not guided by test coverage metrics. Instead, the tool generates input data arbitrarily for the given specification, in order to find bugs. A bug in the context of Hypothesis is an exception being thrown, an assertion being violated or some other basic assumption is being broken. For example an object that does not have the same value anymore after being serialized and then deserialized.

Hypothesis is used with Python decorators. The keyword *@given* followed by a function specifying the type of argument to be used is placed before the function as shown in Listing 2.

[6]HypothesisWorks: Hypothesis testing framework, `https://hypothesis.readthedocs.io/en/latest/`

```python
# returns False if temp is smaller than 1000
def isDangerous(temp):
        if(temp <= 1000): # bug: <= instead of <
                return False
        else:
                return True

# this decorator tells Hypothesis to use integer values as parameters
@given(integers())
def isDangerous_test(temp):
        if(temp < 1000):
                assert isDangerous(temp) == False
        else:
                assert isDangerous(temp) == True
```

Listing 2: An example of how Hypothesis should be used

Hypothesis will now generate random test inputs according to the specification, integers in case of this example. If failing input is found, it is stored in a local database. This way, the falsifying example can be reused in future runs to rule out bugs that were already found earlier.

2.6 Other tools

We used the tool `coverage.py`[7] mainly to find out which lines are not covered by a test suite. As Pynguin incorporates its own line coverage measurement tool, we only used `coverage.py` to evaluate line coverage of manually written test suites.

Our test runner for manual test suites and Hypothesis is `unittest`[8]. We use `unittest` over `pytest`[9] as the original test suite for our SUT uses `unittest`, which we built our test suite upon. We still use `pytest` to run Pynguin-generated test suites.

For mutation testing, we use `mutmut`[10]. The tool can generate mutations to a given program and will evaluate whether the test suite can kill them. `mutmut` thus implements strong mutation coverage.

2.7 Mock Testing

Testing software can be a time-consuming process. This is particularly the case if the testing process involves software artifacts that are not supposed to be tested. A possible way to reduce testing times in such cases is mock testing[4], which means replacing

[7]Documentation of Coverage.py, `https://coverage.readthedocs.io/en/6.2/`

[8]unittest — Unit testing framework — Python 3.10.2 documentation, `https://docs.python.org/3/library/unittest.html`

[9]pytest documantation, `https://docs.pytest.org/en/6.2.x/`

[10]mutmut - python mutation tester `https://mutmut.readthedocs.io/en/latest/`

certain parts of the system under test with so-called mock objects. A mock object has the same API as the object or artifact it replaces, but no actual functionality that needs to be tested. This is useful for reducing testing time, but also for locating faults by taking a part of the system under test out of the testing process.

In more detail, a mock object Obj' replaces an element Obj in the system during testing. Obj has functions $f_1, ..., f_n$ which can be called by other objects. These do not know that Obj was replaced and will interact with Obj' as if it was the original object, for example by calling a function f_i. Rather than checking if f_i works correctly, Obj' can now check if f_i was called with the correct arguments, or if it was even called at all. The mock object does not test the component it replaces, but the context in which it is used.

Mock testing allows faults to be distinguished between ones inside Obj and ones in the way Obj is used. It can however also save testing time. Any expensive calculations that would run in Obj are skipped during testing with Obj'. This is for example useful if Obj is tested separately.

It should however be noted that for integration testing, mock objects should not be used liberally. Integration testing is the practice of testing that different components of a system work together as intended. For that purpose, it is important for individual components to actually react to input they receive and not just return predefined behavior as mock objects do.

2.8 System under test: elastic2

Elastic2 is a part of the ASE (Atomic Simulation Environment)[11] material science framework. It calculates the elasticity of materials, and can determine if a material will break under a certain deformation. Elastic2 is being developed by Daniel Thomas Speckhard at the Fritz Haber Institute of the Max Planck Society.

Elastic2 is a tool for density-functional theory (DFT) and uses calculators, which are also called DFT codes. These are programs that run the actual simulations on an atomic level. Currently, elastic2 supports three calculators *aims*[12], `exiting`[13], and *Quantum ESPRESSO*[14]. Calculators are usually written in FORTRAN to allow greater efficiency in parallelized computations. DFT calculations aim at investigating material properties on an atomic level. Mathematically, these calculations mostly come down to eigenvalue problems.

An ASE Atoms object is required as an input variable for elastic2. It represents a number of atoms and their position relative to each other. It contains additional information, such as if the structure is a cell that repeats itself indefinitely in a certain direction. Elements in an Atoms object can be represented by ordinal numbers (e.g., 6 for carbon) or symbols (e.g., C for carbon). The minimum required to construct a non-empty Atoms object is a string with symbols or an array of numbers. Optional arguments like positions can be provided as well and are required by many functions in elastic2.

Elastic2 is written in Python and can be run via the command line. The workflow for the full release version will look as follows: The user inputs a structure of atoms, a

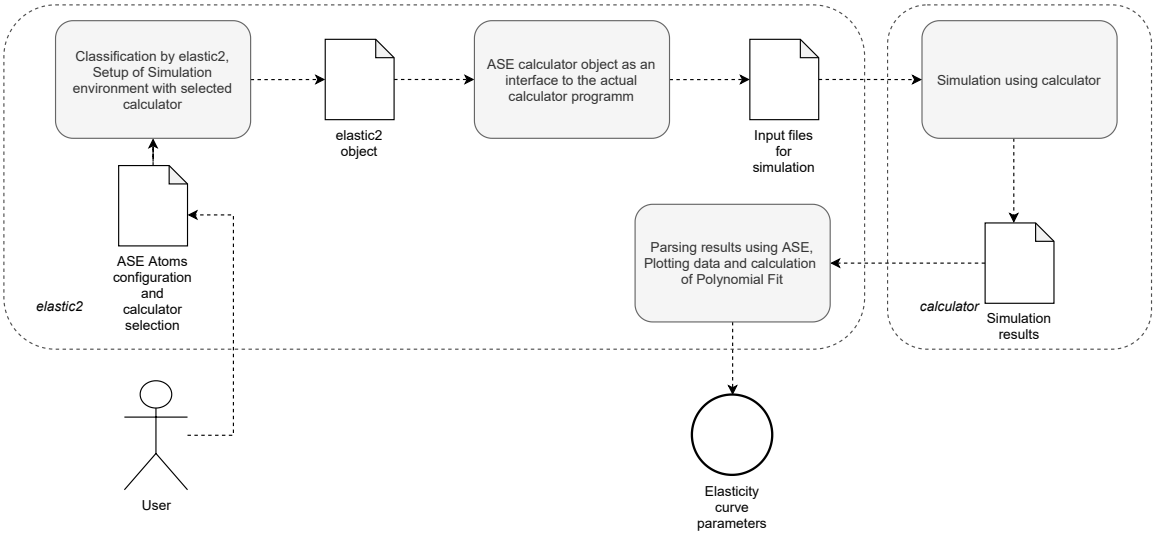

Figure 1: The basic structure of elastic2

deformation which should be simulated, and a calculator name. When run, elastic2 will prepare the input for the simulation in the module **setup_runs**. The calculator is abstracted by an "ASE calculator" object inside elastic2. This interface creates input files for the actual calculator, which then simulates the energy of the material at certain points during the deformation. Once the simulation is finished, the results are parsed by elastic2 using ASE in the module **analyze_runs**. A polynomial curve is formed from the points that were simulated. It represents how energy between the atoms in the structure changes while being deformed. The structure of elastic2 is visualized in figure 1.

We use a slightly altered version[11] of elastic2 for the context of this thesis. It represents the current state of development, however unfinished parts were omitted. In our version, the DFT-code simulation is prepared but not actually run. Also, we created a copy of the module `setup_runs.py`, which does most of the work. This copy is called `setup_runs_safe.py` and has all operations which alter hard drive contents replaced with `print` statements. This is mainly done for safety reasons, to keep the program from writing files at unwanted locations when given random inputs. We also replaced the custom exceptions used by elastic2 with standard exceptions such as `ValueError` or `RuntimeError`, as raising a custom exception would cause a crash in Pynguin.

[11] elastic2 in thesis repository on HU gitlab, `https://scm.cms.hu-berlin.de/trappejo/elast2tstcov/Software/elastic2/`

3 Development and Experiment

In order to answer **RQ1**, we evaluated automatic test generation with different algorithms provided by Pynguin, which are Pynguin's *FD-random*, *whole suite*, *DynaMOSA*, and *MIO*. They are run on the elastic2 module `setup_runs`. We use line coverage as our main metric. Additionally, we use mutation coverage to examine the generated test suites.

We also developed two test suites for `elastic2.setup_runs`. The first one is a "traditional" test suite using Python `unittest`. We incorporated mock functionality into this test suite in order to answer **RQ2**. The second test suite uses the Hypothesis framework, which was introduced in section 2.5.

3.1 The gold standard

In order to assess potential difficulties for test case generation, we developed a test suite for elastic2 with a line coverage of almost 1.0. The only parts it does not cover are running the actual simulation, which is mocked during testing, and lines that cannot be executed successfully currently due to missing dependencies. This test suite is built upon the original test suite developed for elastic2 by its author. The original test suite uses `unittest` and incorporates `pytest` elements, such as parameterized tests. We used the tool `coverage.py` to find lines not covered by the current test suite and we wrote test cases covering those. During the creation of this test suite, we inserted error handling into the system under test in locations that otherwise could have led to crashes during test case generation.

This gold standard integrates mocking. Not an entire object is replaced during testing, but a single function. In our case, this is `setup_runs.run_simulation`, which would start the computationally expensive calculations made by the DFT code. This is trivially the best performing test suite regarding line coverage out of all the ones created in this project. However, the gold standard is also most costly in terms of working hours.

3.2 Test suite using the Hypothesis framework

The Hypothesis-based test suite is essentially a hybrid approach between manual testing and automated test case generation. Hypothesis randomly generates inputs for hand-written tests according to a given specification. This allows more effective random testing, since the developer has direct control over the input that is generated.

Several examples can be found in our SUT where a more precise description of a function argument could greatly reduce testing time. One such example is the name of the DFT code, which is passed to functions as a string. Even with a type hint, other test case generation strategies would of course generate a variety of random strings, of which very few were actually viable DFT code names. In our Hypothesis test suite, we can just tell the tool to pick one of the available DFT code names or use an invalid string to test error handling functionality.

Hypothesis however does not try to achieve a certain coverage, but rather to find inputs that raise errors or break assumptions. The test suite is thus not directly comparable to the ones generated by automated test case generation techniques.

3.3 Experiment: Automated test case generation with Pynguin

We applied four automated test case generation techniques on the elastic2 module `setup_runs_safe.py`. These are *FD-random*, *whole suite*, *DynaMOSA*, and *MIO*. Each of them was given a runtime of 900 seconds, while each algorithm was run 40 times. We chose 40 runs to ensure stability in our results. While we initially opted for a runtime of 600 seconds like in the similar experiment by Lukasczyk et al.[3], we observed unexpected results for the *MIO* algorithm. We wanted to test whether these result would remain the same with a longer run time, which they did. Except for the runtime and the algorithm to use on each run, Pynguin was executed with default parameters.

We used a parallelized configuration with `slurm` to run the experiments. Our experiments were run on a Dell R740xd server with 756GB and two Xeon 6254 CPUs, each with 36 cores and 72 threads, operating at 3.1GHz. It the time of running the experiments, the server ran on openSUSE Leap 15.3 with the kernel version 5.3.18-59.40-preempt. The tools we used and their respective versions are listed in table 1. Detailed instructions on how to reproduce our results can be found in the README file[12] of the thesis repository.

[12]README.md in thesis repository on HU gitlab, `https://scm.cms.hu-berlin.de/trappejo/elast2tstcov/-/blob/main/README.md`

Tool or module	**version**
ase	3.22.1
attrs	21.4.0
coverage	6.2
file_read_backwards	2.0.0
hypothesis	6.31.6
matid	0.6.2
mock	3.0.5
mutmut	2.2.0
numpy	1.19.4
openpyxl	3.0.9
packaging	21.3
pandas	1.4.0
pynguin	0.11.0
pytest	6.2.5
python	3.8.8
typing-extensions	4.0.1

Table 1: Tool and modules used in this work and their respective versions.

4 Evaluation

4.1 Evaluation methodology

Individual line coverage We displayed the results of our Pynguin test case generation runs as box plots, representing how the line coverage for each generation strategy developed over time. The coverage data collected during generation is aggregated in one box plot for all runs of a single algorithm. The box represents the IQR (interquartile range) whereas the whiskers extend up to the highest or down to the lowest value that is inside a 1.5x IQR range from the respective edge of the box. Any results outside the whisker boundaries are represented by dots. A line inside the box represents the median of all measurements for the respective algorithm. Coverage data is displayed in 30 second intervals. As all box plots are about line coverage, higher values can be interpreted as better.

Comparison of the mean line coverage Additionally, we included a line plot comparing the mean line coverage for all test generation strategies in one plot. Each line corresponds to one algorithm, with the coverage data of all runs being used. For this plot, we used measurements from one second intervals. Again, higher values in line coverage are to be interpreted as better.

Statistical evaluation Based on the results of Campos et al.[10], we made assumptions regarding the performance of test case generation strategies, which we would then try to validate. We compared the line coverage after the testing time elapsed. We created the following set of null hypotheses H_0:

- The test suites generated with *whole suite* have an equivalent or lower line coverage than those generated by *FD-random*
- The test suites generated with *DynaMOSA* have an equivalent or lower line coverage than those generated by any other strategy
- The test suites generated with *MIO* have an equivalent or lower line coverage than those generated *FD-random*
- The test suites generated with *MIO* have a lower line coverage than those generated with *whole suite*

Our evaluation aims to reject these null hypotheses in favor of the following set of alternative hypotheses H_1:

- The test suites generated with *whole suite* should have a higher line coverage than those generated by *FD-random*
- The test suites generated with *DynaMOSA* should have a higher line coverage than those generated by any other strategy

- The test suites generated with *MIO* should have a higher line coverage than those generated *FD-random*
- The test suites generated with *MIO* should not have a lower line coverage than those generated with *whole suite*

We used the non-parametric Mann-Whitney U test for evaluations. We choose this method as we cannot assume a normal distribution among our data. We examined some resulting samples and found that they were not normally distributed. The sample for each test case generation algorithm is made up of the final line coverage measurements for each run. This gives us a sample size of 40 per algorithm. We calculate the results using the `scipy` package[13]. We consider a null hypothesis to be rejected if its associated p-values is smaller than 0.05.

Missing lines Aside from the raw performance of the algorithms, we were interested in specific properties of the SUT that might interfere with test case generation. For that purpose, we analyzed which parts of the SUT were covered by the smallest number of test suites. The percentage of executable statements in `elastic2.setup_runs` not covered by any test suite for each algorithm is displayed in table 3. These information greatly helped us to analyze potential problems the test generators were facing.

Mutation analysis In order to assess the sensitivity of our test suites to errors, we use mutation coverage measured by the package `mutmut`. We let the tool analyze each generated test suite and accumulate the data in one box plot per algorithm. It is configured the same way the box plot for line coverage, with the box representing the IQR while the whiskers cover a 1,5x IQR-range each. A higher mutation coverage is regarded as better.

It is to be noted that the test suites generated by the *FD-random* strategy are exceptionally large, with sizes of around 10.000 test cases each. This is because in contrast to the other algorithms we examined, *FD-random* does not take any measures to minimize the test suites. The immense size of the test suites means a comparably long execution time, usually over 60 seconds per mutant for one test suite. For this reason, we decided not to examine mutation coverage on the test suites generated by *FD-random*, as the test suite would have to run once per mutation. If we applied 100 mutations, this would already take at least 100 minutes, and we would have to run this for every of the 40 generated test suites, which we deemed unreasonable.

4.2 Results and Analysis

An overview comparing the median development of line coverage by algorithm is provided by figure 2. The development of line coverage over time for all test suites of each algorithm is displayed in figure 3.

[13]scipy.stats.mannwhitneyu `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html`

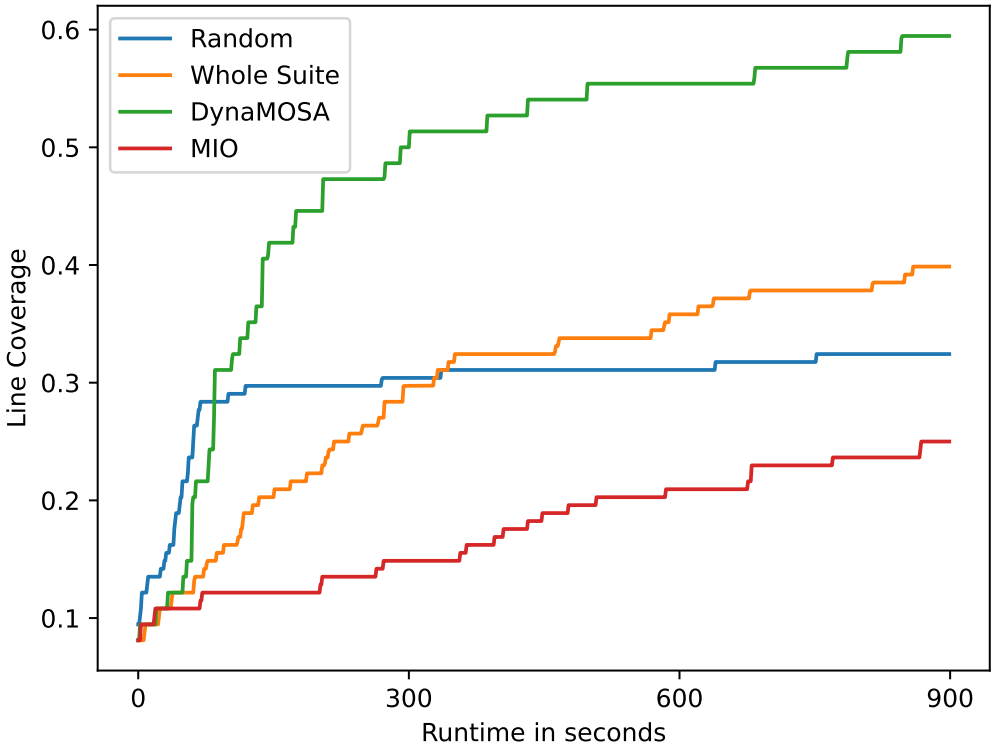

Figure 2: Median line coverage over time in a direct comparison for a runtime of 900 seconds

FD-random The *FD-random* test case generation manages to achieve a comparably good coverage early on. After 30 seconds, it has a higher median coverage than any of the genetic approaches. This can possibly be explained by the overhead which other algorithms have to deal with, for example minimizing their test suites or keeping archives.

The *FD-random* strategy soon loses its lead in median line coverage to *DynaMOSA*. It can be observed that the increase in coverage for *FD-random* gets significantly slower after about 120 seconds. At that point, the median line coverage is at 0.29. In the remaining 780 seconds of runtime, the median will only rise to 0.32. The box plot in figure 3 shows how the measurements for *FD-random* get less diverse towards the end of the generation run, indicating that most generated test suites have achieved a line coverage close to the median value.

As discussed previously, we did not perform a mutation coverage analysis for *FD-random*.

Whole suite The *whole suite* approach slowly but steadily increases its line coverage over time. The median line coverage rises almost in every 30 second step. However,

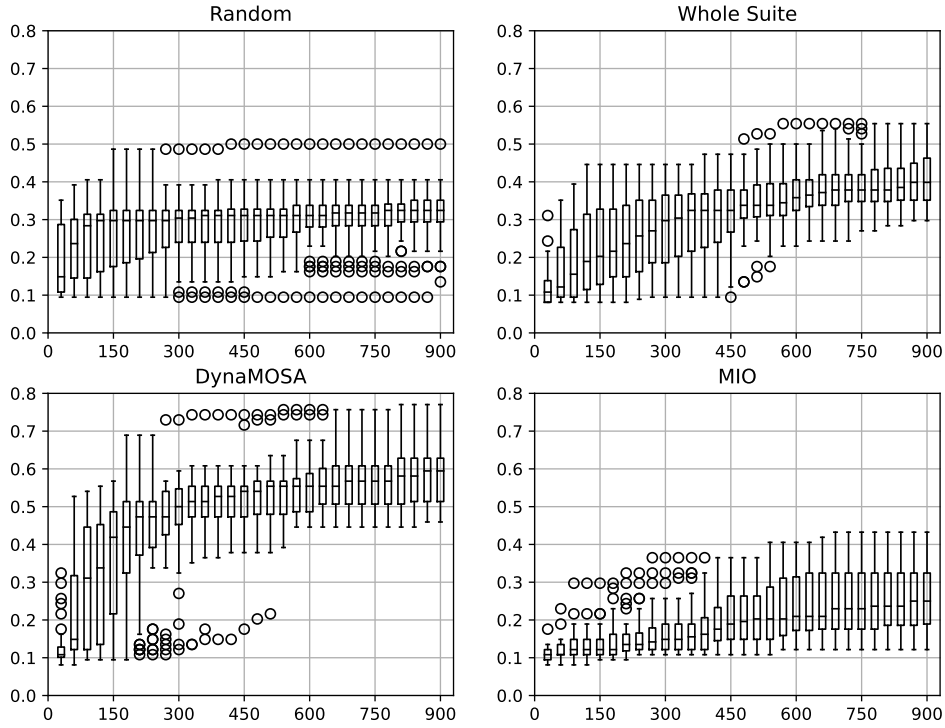

Figure 3: Evolution of line coverage over time for different test case generation algorithms

during the first half of the generation runs, the measurements are spread over a wide range: while some candidates already reached a line coverage of over 0.4 after 120 seconds, at least one test suite was still covering less than 10% of the lines when half the generation time had elapsed.

After 360 seconds, *whole suite* surpassed *FD-random* in median line coverage. At the end of our run, *whole suite* had a median line coverage of 0.39. We evaluated the final measurements of *whole suite* and *FD-random* in the Mann-Whitney U test, with the null hypothesis that the line coverage for *FD-random* is stochastically the same or better than that for *whole suite*. We can reject this hypothesis, as the p-value for it lies at $3.54002363 \cdot 10^{-8}$, which is smaller than our significance level of 0.05.

In regards to mutation coverage however, *whole suite* performed very poorly as seen in table 4. To be exact, no test suite generated by the *whole suite* was able to kill a single mutant. We first assumed this to be an error, but by looking closer into the test suites, we uncovered a possible reason for these results.

As previously mentioned, Pynguin generates two files of test cases: one file for test cases that succeed, and one for failing test cases. For *whole suite*, the succeeding test case file was empty on most occasions. If it did contain a test case, it only contained a

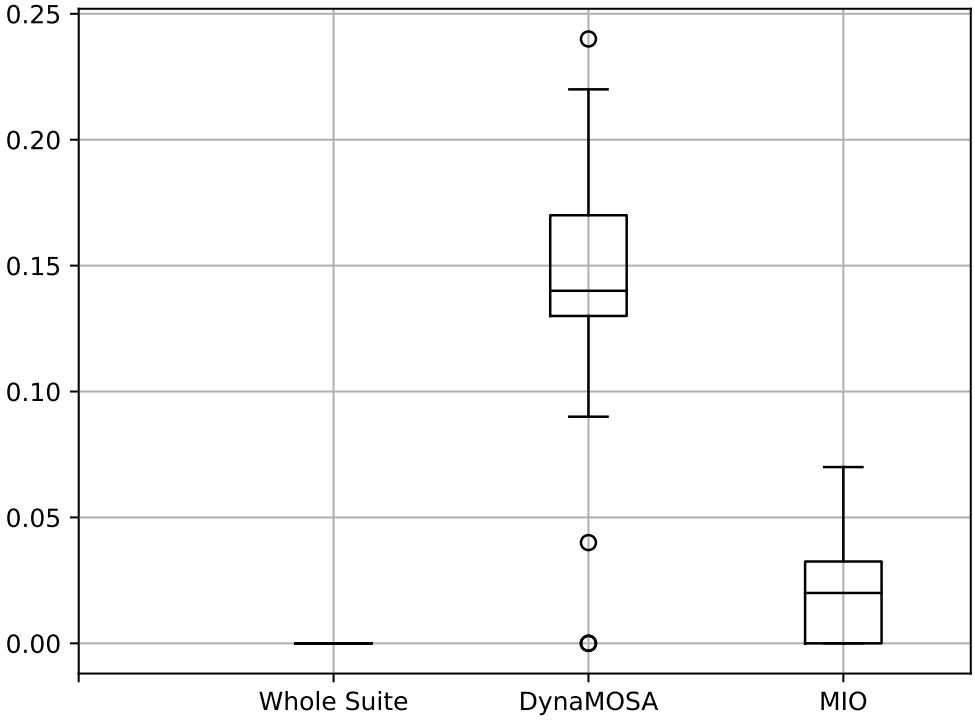

Figure 4: Mutation coverage of the final test suites, by algorithm

call to `get_dft_calculator`, which is the only static method in our SUT.

How is this related to a bad mutation coverage? First of all, we see that most coverage *whole suite* achieves comes from failing test cases. When Pynguin generates a test case and runs into an error, it might still add the test case to the test suite if it helps to meet the coverage goals of an algorithm like *whole suite*. The test case is then wrapped in a `try except` statement.

Our mutation coverage tool considers a mutant killed when it gets a non-zero return value from the test case execution. This will never happen for failing test cases, as errors are caught in the `try except` statement without altering the return value of the testing run. As for why `get_dft_calculator` was apparently never mutated in a way that would have been discovered, the explanation could be an issue in `mutmut`, which is discussed in section 4.4.

In conclusion, we can say that *whole suite* was able to achieve a significantly higher line coverage than *FD-random*, the generated test suites are however of little practical value due to them mainly consisting of failing test cases. While those can be investigated individually to find potential faults, they are not sensitive to new errors as proven by the non-existent mutation coverage.

H_0	H_1	**p-value for H_0**
FD-random $\geq$ Whole Suite	FD-random $<$ Whole Suite	$\mathbf{3.54002363 \cdot 10^{-8}}$
FD-random $\geq$ DynaMOSA	FD-random $<$ DynaMOSA	$\mathbf{1.6317291 \cdot 10^{-14}}$
Whole suite $\geq$ DynaMOSA	Whole suite $<$ DynaMOSA	$\mathbf{5.73774652 \cdot 10^{-12}}$
MIO $\geq$ DynaMOSA	MIO $<$ DynaMOSA	$\mathbf{9.71077652 \cdot 10^{-15}}$
FD-random $\geq$ MIO	FD-random $<$ MIO	0.9984576
Whole suite $\geq$ MIO	Whole suite $<$ MIO	1.0

Table 2: Statistical evaluations of our assumptions regarding line coverage. The values that are small enough to reject H_0 are marked in **bold**.

algorithm	**number of missed lines**	**miss rate**
FD-random	53/195	27.18%
Whole Suite	48/195	24.62%
DynaMOSA	26/195	13.33%
MIO	57/195	29.23%
Combined	25/195	12.82%

Table 3: How many lines were missed by each test case generator, meaning how many executable statements were not covered by any test suite for the respective algorithm. The row "Combined" shows the number of lines not covered by any of the automatically generated test suites.

DynaMOSA *DynaMOSA* vastly outperformed any other strategy we examined. From the 90 second mark, its median line coverage exceeded that of all other algorithms. At the end of the generation run, the median line coverage of *DynaMOSA* was at 0.59, which is higher than the maximum line coverage achieved by any run an another algorithm over the entire runtime.

The Mann-Whitney U test supports the assumption we made earlier, that *DynaMOSA* test suites would have a higher line coverage than any other algorithm. Detailed results can be found in table 2. It is also no surprise that *DynaMOSA* has the lowest amount of missed lines out of all algorithms, as shown in table 3. A single line of the SUT (line 237) is not covered by *DynaMOSA*, but is covered by *FD-random*. All other lines executed by any generated test suite are also covered by at least one *DynaMOSA* test suite.

DynaMOSA also generated more failing than passing test cases. It did however achieve a higher mutation coverage than the other two genetic algorithms, with a median mutation coverage of 0.14. Based on this, we conclude that *DynaMOSA* generated test suites would be the most sensitive created in this experiment.

By all metrics we used, *DynaMOSA* outperformed other automated test case generation techniques. It had the highest median line coverage, the highest absolute line coverage, the lowest amount of missed lines, and the best mutation coverage. This aligns with results of previous experiments.

MIO The greatest surprise to us when seeing the results was the bad performance of *MIO*-generated test suites. Not only were they consistently outperformed by the other genetic algorithms, they were not even able to compete with test suites generated by *FD-random*.

In a previous run, our experiment was given a runtime budget of 600 seconds. After seeing the poor performance of *MIO*, we decided to increase the time by 50%, which however did not change our results.

MIO has performed better in other studies, such as the one conducted by Campos et al.[10] or the evaluation by the authors of *MIO*[9]. We can only speculate why it performed so poorly in our evaluation. The SUT might inhibit specific properties that prevent *MIO* from functioning as intended. *MIO* in Pynguin is configured to start its focus phase after 50% of the testing time has elapsed. This could be too early.

Even though it has a lower line coverage than *whole suite*, *MIO* does have a higher mutation coverage. With a median of 0.02 and a maximum of 0.07, it is still far worse than *DynaMOSA*, but that is to be expected with the much lower line coverage it achieves. If a line is not executed by a test suite, a change to it is certainly not detected.

In a recent study[15] conducted by Lukasczyk et al. with Pynguin, more advanced algorithms were compared, namely MOSA, *DynaMOSA*, and *MIO*. Those were tested on a total of 20 Python projects, ranging from 89 to 3311 lines of code in size. Regarding median and mean performance, *MIO* was slightly behind *DynaMOSA* and *MOSA*, while all three performed better than *whole suite* and *FD-random*. Simply due to the much larger code base examined in their study, we do not consider our findings regarding *MIO* as universally valid and regard them as an outlier. Why exactly *MIO* performed poorly on our SUT is an open question for further research.

MIO performed much worse than we expected regarding line coverage. The *MIO* test suites appear to still be more sensitive than those generated by *whole suite*.

RQ1: We were able to confirm performance expectations in relation to each other for *FD-random*, *whole suite* and *DynaMOSA* in regards to line coverage. *MIO* performed far worse than we had anticipated, but we do not consider this to be universally valid for test case generation Python.

Automated test case generation and safety While running our first experiments with Pynguin, we noticed lots of obscurely directories that were created while generating tests for our SUT. This happened because elastic2 has functions that can create directories and files. The authors of Pynguin are aware of such possibilities, and therefore force users to set a certain flag before being able to run Pynguin. This flag shall remind users that the system under test is executed during test generation and can potentially damage the system it is running on. For the same reason, we created a safe version of our SUT. To mitigate potentially unsafe behavior, another concept already discussed in this thesis could be used: mock testing. Python already offers functionality[14] to mock its `open` function, which is used to open files. An optional future integration of mocking into test generation might help reduce safety risks.

In the manual test suite, we mitigated safety concerns by incorporating mock testing to have better control over file writing operations. We also hope to be able to reduce testing time in future versions of the SUT by using mock testing. We were however unable to conduct experiments on this, as elastic2 is currently not capable of running actual simulations.

RQ2: Mock objects helped us to improve safety during testing. We pointed out a possible use case of mock testing in combination with automated test case generation, which should be explored in future research.

4.3 Challenges for automated test case generation

In this section, we will discuss possible reasons for problems test case generators could have faced when trying to generate tests for our SUT. To do this, we analyzed lines that were missed by all test suites. The function `setup_runs()` calls a number of other functions to prepare the inputs for the simulation using a DFT code. At first, `setup_runs()` will perform a number of checks to the Atoms object, such as if the object contains multiple Atoms, which is required. One of them is performed in `determine_xtal_structure()`: the Atoms object has to be verified as a valid three-dimensional structure.

Generating such an object is very difficult, which is why the lines executed after a successful classification in `determine_xtal_structure()` are not reached by any test suite. However, this also means that any line after the call of `determine_xtal_structure()` in `setup_runs()` will not be reached, even if the test generators have already covered the bodies of the functions that are called there.

Some functions in the SUT require others to be called before in order to reach certain parts of the code. These implicit dependencies can be hard to resolve for test case generators that mainly observe how changes affect the coverage goals they try to achieve. The lack of type hints in some function arguments can also be challenging for

[14] mock_open Documentation, `https://docs.python.org/3/library/unittest.mock.html#mock-open`

test case generation. Two highlight both of these difficulties, consider the example in Listing 3, which is taken from elastic2 and had comments and empty lines removed.

```
1  def convert_voigt_to_mat(self, strain_vec):
2          eta_matrix = np.zeros((3, 3))
3          eta_matrix[0, 0] = strain_vec[0]
4          eta_matrix[0, 1] = strain_vec[5] / 2
5          eta_matrix[0, 2] = strain_vec[4] / 2
6          eta_matrix[1, 0] = strain_vec[5] / 2
7          eta_matrix[1, 1] = strain_vec[1]
8          eta_matrix[1, 2] = strain_vec[3] / 2
9          eta_matrix[2, 0] = strain_vec[4] / 2
10         eta_matrix[2, 1] = strain_vec[3] / 2
11         eta_matrix[2, 2] = strain_vec[2]
12
13         return eta_matrix
14
15 def get_deform_matrix(self, strain_coeff: float, strain_vec):
16         distortion_vec = np.multiply(strain_coeff, strain_vec)
17         eta_matrix = self.convert_voigt_to_mat(distortion_vec)
18         ...
```

Listing 3: elastic2 code fragment which shows the relevance of type information.

The argument `strain_vector` of `get_deform_matrix` has no type hint, thus test case generators can insert any randomly generated variable or object for it. To not throw an exception, the operation in line 16 requires `strain_vector` to be a valid input for `numpy.multiply`, which 33.3% of our generated test suites managed to achieve. However, to successfully execute the next line, which calls `convert_voigt_to_mat`, `strain_vector` has to be a vector with at least 6 elements, most of which need be of a data type that can be divided by an integer. Only 6.9% of our generated test suites were able to generate an input that would not result in an error at line 17. This is particularly interesting because 53.5% of all test suites were able to generate valid inputs for `convert_voigt_to_mat`, but were apparently unable to put them in the right place. As a result not only line 17, but the entire rest of the function, which includes 13 more statements, are covered by only very few test suites.

4.4 Threats to validity

Randomness in test case generation We evaluated test case generation algorithms which by their nature rely on (pseudo-)randomness. This is especially true for the *FD-random* test case generation with Pynguin or Hypothesis, but also for the evolutionary algorithms *whole suite*, *DynaMOSA*, and *MIO*. The latter ones randomly generate initial test suites and modifications, like mutation and crossover, are applied based on

random numbers. We tried to mitigate this be running 40 runs per generator, but we cannot fully eliminate the possibility of our results being influenced by randomness.

The SUT Our experiments were carried out on a single module with 241 executable lines of code. Compared to other experiments, like the ones conducted by Lukasczyk et al.[3][15], this is not a large code base. Certain properties of the SUT, such as its reliance on a fairly complex object as an input in form of ASE Atoms, could have favored some generation techniques while putting others at a disadvantage. The bad performance of *MIO* might be a hint for this being the case.

Issue with mutmut The mutation testing tool `mutmut` used by us identified a total of 238 mutations it would apply on our SUT in one run, according to its console output. It would however stop generating mutants after it had generated exactly 100. This is apparently a known issue[15] and has to our knowledge not been fixed yet. We decide to use the achieved results anyway since 100 mutants were consistently generated on each run, but our results are likely influenced by this issue.

Different tools We used `unittest` as our test runner for the manual test suite, while Pynguin uses `pytest`. Additionally, Pynguin has an own implementation to measure coverage. We used `coverage.py` for our manual test suite. Differences in these tools might have influenced our results, and should be taken into account when comparing coverage measurements.

[15]mutmut issues: Mutmut stops running #183, `https://github.com/boxed/mutmut/issues/183`

5 Conclusion and Future Work

While there might not be as many testing tools as there are in other languages, there are still considerable possibilities for testing in Python. In this thesis, we presented different tools and applied them to a real world program. We implemented a manual and a semi-automatic test suite for the system under test. Furthermore, we conducted an experiment to compare different test case generation algorithms in Python. Some expectations set by other languages could be confirmed by our results, such as the generally good performance of *DynaMOSA*. Other findings do not line up with current research, notably the bad performance of *MIO*. We took a closer look at the results of our experiment and analyzed possible challenges our system under test provides to automated test case generation.

One topic for future research is the unexpectedly bad performance of the *MIO* algorithm in our experiment. The possibility of combining concepts discussed in this thesis, namely test case generation and mock testing, is also a subject of future research.

References

[1] Gordon Fraser and Andrea Arcuri. A.: Evosuite: Automatic test suite generation for object-oriented software. In *In: Proc. of ACM SIGSOFT ESEC/FSE*, pages 416–419, 2011.

[2] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007.

[3] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. *Lecture Notes in Computer Science*, page 9–24, 2020.

[4] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2000.

[5] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[6] Melanie Mitchell. *An Introduction to Genetic Algorithms*. 1996.

[7] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[8] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, February 2018.

[9] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. *Lecture Notes in Computer Science*, page 3–17, 2017.

[10] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for test suite generation. In Tim Menzies and Justyna Petke, editors, *Proceedings of the 9th International Symposium Search-Based Software Engineering (SSBSE)*, pages 33–48. Springer International Publishing, Cham, 2017.

[11] Ask Hjorth Larsen, Jens Jørgen Mortensen, Jakob Blomqvist, Ivano E Castelli, Rune Christensen, Marcin Dułak, Jesper Friis, Michael N Groves, Bjørk Hammer, Cory Hargus, et al. The atomic simulation environment—a python library for working with atoms. *Journal of Physics: Condensed Matter*, 29(27):273002, 2017.

[12] Volker Blum, Ralf Gehrke, Felix Hanke, Paula Havu, Ville Havu, Xinguo Ren, Karsten Reuter, and Matthias Scheffler. Ab initio molecular simulations with numeric atom-centered orbitals. *Computer Physics Communications*, 180(11):2175–2196, 2009.

[13] Andris Gulans, Stefan Kontur, Christian Meisenbichler, Dmitrii Nabok, Pasquale Pavone, Santiago Rigamonti, Stephan Sagmeister, Ute Werner, and Claudia Draxl. Exciting: A full-potential all-electron package implementing density-functional theory and many-body perturbation theory. *Journal of Physics: Condensed Matter*, 26:363202, 08 2014.

[14] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, and et al. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502, Sep 2009.

[15] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for python, 2021.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 2. Februar 2022 .....