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

Daniel Trübenbach
Humboldt-Universität zu Berlin
Germany

Sebastian Müller
muelerse@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

Lars Grunske
Humboldt-Universität zu Berlin
Germany

ABSTRACT

Writing software tests is essential to ensure a high quality of the software project under test. However, writing tests manually is time consuming and expensive. Especially in research fields of the natural sciences, scientists do not have a formal education in software engineering. Thus, automatic test case generation is particularly promising to help build good test suites.

In this case study, we investigate the efficacy of automated test case generation approaches for the Python project *Atomic Simulation Environment* (ASE) used in the material sciences. We compare the branch and mutation coverages reached by both the automatic approaches, as well as a manually created test suite. Finally, we statistically evaluate the measured coverages by each approach against those reached by any of the other approaches.

We find that while all evaluated approaches are able to improve upon the original test suite of ASE, none of the automated test case generation algorithms manage to come close to the coverages reached by the manually created test suite. We hypothesize this may be due to the fact that none of the employed test case generation approaches were developed to work on complex structured inputs. Thus, we conclude that more work may be needed if automated test case generation is used on software that requires this type of input.

KEYWORDS

Search Based Testing, Test Case Generation, Scientific Computing, Scientific Software

ACM Reference Format:
Daniel Trübenbach, Sebastian Müller, and Lars Grunske. 2022. 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. In *The 15th Search-Based Software Testing Workshop (SBST'22), May 9, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3526072.3527523

SBST'22 , May 9, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9318-8/22/05.
https://doi.org/10.1145/3526072.3527523

1 INTRODUCTION

With ever-growing software projects, testing software is becoming increasingly important to assure its quality [24]. As writing tests manually is costly, automating the generation of tests by use of metaheuristic search techniques is an active field of research [22, 24, 28]. For the statically-typed language Java, *EvoSuite* [7, 8] is a popular example for such a search-based approach to automated *test case generation (TCG).* It has been shown that EvoSuite is practically relevant by producing tests that achieve "good levels" of code coverage [9, p. 1]. The EvoSuite-produced tests also detect real faults in open-source software [10]. *Pynguin* is an automated unit test generation tool for the dynamically typed language Python [19] that provides baseline TCG algorithms such as *Whole Test Suite* generation [8]. The prototype tool is still under active development and state-of-the-art TCG algorithms such as DynaMOSA [27] and MIO [1] were added to Pynguin [20] recently. Thus, *Pynguin* now provides many state-of-the-art algorithms also available in the more mature tool *Evosuite.*

In this case study, we investigate the efficacy of automated TCG compared to the manual creation of a test suite for the workflow system *Atomic Simulation Environment* (ASE) [17] used in computational material science. Within ASE, physical properties of molecular structures are simulated: If a material shows promising results in the simulations, that material may be synthesized and analyzed in wet-lab experiments [29]. ASE employs several different Density Functional Theory (DFT) [13] codes, that may need to be wrapped by a so-called *Calculator* for compatibility reasons. While the DFT codes are usually written in Fortran, ASE itself is a Python project. In this paper, we investigate the *Calculator* for the DFT code called `exciting` [11]. We introduce ASE in more detail in section 2.2.

Open Challenges. The interface between ASE and the `exciting` *Calculator* is still under active development. Certain challenges exist in scientific software that pose a problem to the software quality assurance: (i) the expected output may not be easily predicted [30]; (ii) often, such software is developed by multiple people over long timeframes while neglecting code documentation [16]; as well as (iii) missing education in methods of software engineering by the scientist developing the research software [15]. A low test coverage may lead to problems in long-term extensibility and maintainability [12, 29]. A lasting improvement in these quality measures may only be seen, if testing techniques are integrated into the development workflows [14]. Therefore, tools for TCG may be a help in quality

assurance in the context of scientific software. However, previous work by Lim et al. has shown that human written tests are better at avoiding overfitting to the software under test [18]. We thus ask the following, guiding research question:

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?

The main contributions of this paper are:

- Statistically evaluating the efficacy of different TCG algorithms within the *Pynguin* tool in the context of the `exciting` *Calculator* in ASE.
- Comparing the coverage of automated generated test suites using the TCG algorithms of *Pynguin* with a manually created gold standard.

2 BACKGROUND

2.1 Automatic Test Case Generation

The goal of automatic test case generation (or TCG for short) is to generate a test suite without interacting with a user while still reaching high coverage metrics. Thus, TCG can be viewed as an optimization problem, where the coverage metrics (such as line, branch, or mutation coverage) are the criteria to be optimized using a limited time or computation budget [1]. In statically typed programming languages, automatic test case generation is an active field of research. Multiple approaches to generating test suites reaching high values in the different coverage metrics are implemented in mature research prototypes such as EvoSuite [7]. Recently, a research prototype called Pynguin has been developed [19, 20] to bring the state-of-the-art approaches to the dynamically typed language of Python.

Mitigating the threat that TCG algorithms produce very large test suites without specific oracles, Fraser et al. proposed the *Whole Test Suite Generation* approach [8]. The approach not only tries to reach a high coverage, but also tries to minimize the size of the test suite. To that end, Fraser et al. implemented a genetic algorithm.

The *Many-Objective Sorting Algorithm* (MOSA) [27] is an adaptation of the *Whole Test Suite Generation* approach. It changes how test cases are selected, so that test cases are preferred that are closest to fulfilling a yet uncovered goal. Should two tests have the same fitness value, the algorithm selects for the next generation the shorter of the two test cases. While MOSA assumes that all statements (i.e. targets to cover) of the class under test are independent from one another, this may not be true in all programs. In fact, an improvement of MOSA is DynaMOSA (*Dynamic Many-Objective Sorting Algorithm*) that only computes test cases for targets that can be immediately reached [27]. This means, that all branches that are still nested beneath other – yet uncovered control flow nodes – are temporarily ignored. *DynaMOSA* thus is able to reduce the number of targets that need to be covered at any one time. It thereby decreases computational complexity. *DynaMOSA* subsumes its predecessor – and it is easy to see that *DynaMOSA* is at least as good as its base variant.

Many Independent Objectives (MIO) [1] is another state-of-the-art algorithm for automatic TCG. For each target to cover, a set of test candidates is created during the search. If that set is filled to a given capacity, the algorithm removes the worst test case in that set and replaces it with the next better one that is encountered during search. Similar to DynaMOSA, MIO selects the shorter of two test cases, should these two test cases have the same fitness value. Generating a new test case can either be done through random selection or by mutation of an existing one. The selection of candidates that are to be mutated is done by the number of previous mutations. A high such number shows, that no improvements were found to cover the target, since otherwise the test case would have been replaced. When a target is reached, all other test case candidates for this target are discarded. To cover as many targets as possible within the given budget, the probability to generate a new random test, as well as the size of the test case candidate set are adapted [1]. It was shown that *MIO* can, on average, cover more than *Whole Test Suite Generation* and in some cases is also better than *MOSA* [1].

All three algorithms mentioned above are implemented in Pynguin [20]. Additionally, Pynguin provides an implementation of random feedback-directed generation, similarly to the Randoop approach in Java [25]. All four of approaches are non-deterministic.

2.2 System under Test

Atomic Simulation Environment (ASE) [17] is a Python software package of approximately 67.5k LOC, that provides an environment to calculate as well as simulate physical properties of molecular structures. In ASE, the molecular structure is stored as a geometry of chosen atoms in the well-defined *ASE Atoms Object*. While the ASE Atoms Object is a unified data format, the so-called *DFT codes* used for simulating the physical properties usually use their very own data format for inputs, as well as their outputs. Therefore, ASE employs the GOF adapter pattern to create compatibility between the Atoms Object and the simulation codes: Each code is wrapped by a so-called *Calculator* that translates between the data formats. Figure 1 shows an abstract overview of the workflow encoded in ASE.

All of the codes used within ASE are based on Density Functional Theory (DFT) – a quantum mechanical theory that uses electron density calculations to derive physical properties [13].

`exciting` [11] is such a DFT code for material simulation. We have access to the developers of the DFT code, and as of March 14, 2022 `exciting` is used 24,799 times within the NOMAD repository[1]. Therefore, it is an important code within the material science community and convenient for us to choose due to the access to its developers. `exciting` is written in Fortran 90. Input for `exciting` simulations is given through a *.xml file. The input must follow a defined syntax so that all information to the molecular structure and calculation parameters can be found and read by `exciting`'s internal parser. The simulation's resulting output is also saved in a *.xml file. Thus `exciting` is one of the codes that requires a calculator in order to run correctly. Additionally, the *Calculator* decides if a simulation using the actual DFT code is necessary, as each simulation is very costly in terms of computation time as well as resources.

[1] https://nomad-lab.eu/prod/rae/gui/search?visualization=method

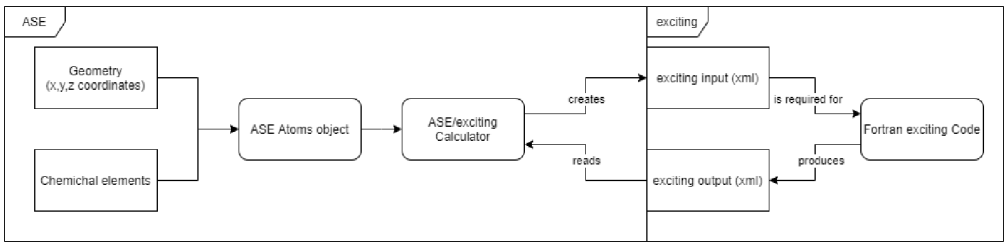


Figure 1: Abstract overview of the workflow within the ASE software.

3 EXPERIMENTAL SETUP

For Python projects, *pytest*[2] is the de-facto standard unit testing framework. Thus, many tools are compatible with pytest by default. Similarly to unit test frameworks in other languages, pytest provides an evaluation of successful and failed test cases after their execution. The framework is written to be extendable and we make use of plug-ins for pytest such as *pytest-cov*[3]. This plug-in adds measuring line and branch coverage to the pytest testing framework.

Cosmic-Ray[4] is a tool that may be used for mutation testing in Python. It generates mutants of the program under test and checks whether the test suite is able to kill these mutants. It supports 14 different mutation operations and is able to mark some equivalent mutants [6].

Table 2 shows the tools we used with their respective versions. The DFT code `exciting` uses Fortran 90; all other tools and algorithms we used are written in Python 3.8.11. We ran our experiments on a *Dell R920* compute server, running *openSUSE Leap* version 15.2 (Linux kernel 5.3). The server is fitted with four *Intel Xeon CPU E7-4880 v2 @ 2.5GHz* and thus has 60 physical cores. The server has 1TB of main memory.

In our experiments, we employ the branch coverage, as well as the mutation coverage to gain insights into the efficacy of the four automatic TCG algorithms mentioned above. We also use these metrics to compare the selected algorithms to a manually created test suite. This manually created test suite was developed by the first author in just under 8 hours of work. We had access to the code for the test suite generation, and thus also had direct access to the branch coverage reached by our test suite.

First, we calculate the coverage metrics for both the baseline and the manually created gold standard test suites. As neither of the two changes throughout the remainder of the experiments, we only need to calculate the coverage metrics once for each. However, for the automated TCG algorithms, we need to account for the randomness introduced through mutation and selection of test case candidates. Therefore, we run each algorithm 50 times. We set the search budget for all TCG algorithms to 600 seconds. We used 600 seconds, as preliminary experimentation showed that all four of the selected TCG algorithms reached a plateau in their coverage metrics by then. For all remaining parameters, we use the default values provided by Pynguin.[5]

[2]https://docs.pytest.org/en/6.2.x
[3]https://github.com/pytest-dev/pytest-cov
[4]https://github.com/sixty-north/cosmic-ray
[5]A full replication package is available at https://doi.org/10.5281/zenodo.6364844

Within each of the 50 experimental runs for each of the TCG algorithms, first *Pynguin* is tasked with generating the test suite. For each of the algorithms, we save to disk data that was selected through the *pynguin_configuration.py* file. Then *Cosmic-Ray* is employed to calculate the mutation coverage for the current algorithm and run. And finally, we calculate branch coverage using the *pytest* plug-in *pytest-cov*.

Tool	Version
Python	3.8.11
pytest	6.2.4
pytest-cov	2.12.1
pynguin	0.9.2
cosmic-ray	8.3.5

Figure 2: All tools and their versions as employed in this study.

4 EXPERIMENTAL RESULTS

We use two coverage metrics to evaluate the efficacy of the employed automatic TCG algorithms: Branch and mutation coverage. These metrics are, by definition, values between 0 and 1. As for all generally used coverage metrics, numerically higher values indicate better results. A low variance would be preferable: A high variance points at the fact that the resulting test suites are widely dissimilar for many runs of the same evaluated algorithm.

In the remainder of this paper, *FD-random* refers to the *feedback-directed* random generation algorithm [26] implementation provided by *Pynguin*. With *Baseline* we refer to the original test suite that ASE was provided with, while we use *Manual* to refer to our manually created test suite. The remaining algorithms will be referred to as *DynaMOSA*, *MIO*, and *Whole Suite*.

4.1 Baseline results

Table 1 shows the results of both branch and mutation coverage. The original test suite of ASE only consists of a single test case. The test case manages to reach a line coverage of 0.31, a branch coverage of 0.25, and a mutation coverage of about 0.03. None of the authors had any prior experience with ASE. We did, however, have direct access to one of the developers from the material science field.

To support the physicists developing ASE, we manually created a full test suite. The manually generated test suite consists of 31 test cases consisting of 494 total lines of code. Subtracting lines that only contain imports or helper functions the manual test suite

		Baseline	Manual	MIO	DynaMOSA	Whole Suite	FD-random
Branch coverage	Min	0.2500	1.0000	0.4941	0.4941	0.4588	0.3294
	Avg	0.2500	1.0000	0.5708	0.5494	0.5438	0.4428
	Max	0.2500	1.0000	0.5882	0.6118	0.5882	0.5529
	Var	0.0000	0.0000	0.0004	0.0014	0.0011	0.0026
Mutation coverage	Min	0.0309	0.9057	0.0350	0.0027	0.0000	0.0350
	Avg	0.0309	0.9057	0.1174	0.0960	0.0196	0.0840
	Max	0.0309	0.9057	0.1968	0.1995	0.1698	0.1779
	Var	0.0000	0.0000	0.0030	0.0027	0.0009	0.0014

Table 1: Resulting minimum, average, and maximum coverages, as well as variance per coverage for each approach. Rounded to four significant figures. For the automated TCG algorithms the search budget is set to 600s.

had on average 12.93 lines of code per test. We kept the tests small but numerous so that localizing root causes for any failure is easier. As a further aid, we paid particular attention to using meaningful names for any variables. Each test case in the manually created test suite also comes with a short *Docstring* explaining the test's goal. Using this manually created test suite, both line and branch coverage reach the maximum value of 1.0. The mutation coverage reaches about 0.91. Neither the baseline nor our manually created test suite show any variance as the test suites themselves do not change.

4.2 Results of the automatic TCG algorithms

In Table 1, we also show the measurements for all four automatic TCG algorithms after a runtime of 600 seconds. Here, we see an improvement over the baseline test suite. The best value overall for the automated methods in the branch coverage metric was reached by DynaMOSA with approx. 0.61. However, we can also see that on average (over all 50 runs) MIO was slightly better than DynaMOSA (approx. 0.57 vs. 0.55). In the mutation coverage metric, the automated methods reached a maximum value of just under 0.20, again using DynaMOSA. In this metric as well, we note that on average, however, MIO reaches better results when compared to DynaMOSA (approx. 0.117 vs. 0.096). For DynaMOSA a total of 976 tests were generated across 50 runs. The average DynaMOSA test suite consists of 19.52 tests, that on average consists of 12.2 lines of code. MIO generated a total of 2,426 test cases for an average of 48.52 per suite. Here, the average test case is 8.25 lines long. Whole Suite generated 1,017 test cases across its 50 runs. Thus, Whole Suite generated 20.35 tests per suite on average, that were on average 14.65 lines long. Finally, FD-random generated a total of 79,047 tests across 50 runs. The average test suite generated by FD-random contained 1,580.9 test cases. Here, the average test case contains 38.86 lines of code.

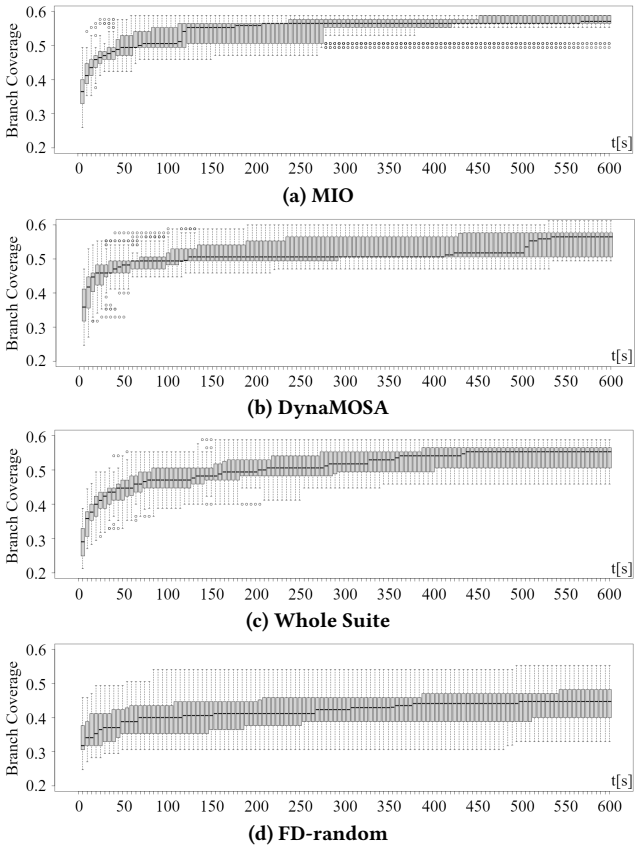
Figure 3[6] shows the evolution of the branch coverage metric over time for each of the four automated TCG algorithms: MIO, DynaMOSA, Whole Suite Generation, and Feedback-Directed random generation. In the boxplots, we present reached branch coverages every 5 seconds. This coverage data is aggregated across all 50 experimental runs per algorithm. The box represents the interquartile range (IQR), while the whiskers extend up to values that are at most 1.5x IQR away from the box boundaries. Any outliers not within any of those boundaries are denoted by the circles. The line within the boxes represent the median value of the population.

We can see that 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. All four methods manage to improve upon the initial random test suite. However, we note that the algorithms show different speeds of improvement: While FD-random generation shows the slowest increase in coverage over time, the state-of-the-art algorithms MIO and DynaMOSA improve quickly at the beginning. However, after the initial phase of fast increases in coverage, the improvement rate slows down for both algorithms. 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.

Furthermore, we note that the variance in the coverage metrics tracked is high for all four algorithms. 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. This effect is especially noticeable in the second phase of the MIO algorithm. Here, fewer random test cases are being generated, but instead more of the existing ones are mutated. This is done to reach targets that are already *close* to being covered. While this behavior allows the algorithm to focus on almost-covered targets within the remaining search budget, it also means that almost no new areas in the program may be found in this phase. It is easy to see that this behavior leads to a decrease in variance.

These general experimental results are inline with data describing the behavior of the selected algorithms in the literature. Studies by Arcuri [1], Panichella et al. [27], as well as Fraser et al. [8], have shown that the TCG algorithms can be ordered by their efficacy: MIO should reach the highest values, DynaMOSA should not be worse than Whole Suite Generation, and all three should be better than FD-random TCG. We can confirm these findings here.

[6] All figures are available as higher resolution images in the online resources accompanying the paper: https://doi.org/10.5281/zenodo.6364844



(a) MIO

(b) DynaMOSA

(c) Whole Suite

(d) FD-random

Figure 3: Branch coverage evolution over time.

4.3 Test Case Readability

```
def test_case_6():
var0 = module0.Exciting()
assert var0 is not None
var1 = {}
var2 = var0.dict_to_xml(var1, var0)
assert var2 is None
```

Figure 4: Example of a MIO generated test case.

In previous studies, Daka et al. [3–5] showed the need for readable test cases in general, as well as in automated test case generation in particular. In her doctoral thesis, Daka argues for the need for further research into unit test name improvement, as well as generating unit tests that use descriptive names for variables [2]. We can affirm these findings through the following anecdotal evidence: We were explaining the test cases to the original developers and users of `exiting`. During that work, we had particular difficulties to both understand, as well as explain the generated test cases fully.

As we can see in Figure 4, the generated test case uses consecutive numbering in naming methods, imported modules, variables, etc. Compared to a manually created test suite where meaningful variable names have been used, we lose contextual clues from this numerical naming scheme that would help the tester to locate exactly what went wrong if a test case failed. Therefore, further work is needed to investigate how generating useful variable names may improve the understanding of developers for the generated test cases and thus may lead to an increase in use of automated TCG tools.

> We can confirm that automatic TCG algorithms are able to improve upon the baseline test suite. We must, however, note that a manually created test suite reaches the maximum value possible for the branch coverage metric and gets close to the maximum in the mutation coverage. Furthermore, the manually created test suite provides the developers with more feedback through meaningful naming schemes that the automatic TCG algorithms are not able to match.

5 ANALYSIS

For all statistical tests in this section, we used the non-parametric Mann-Whitney-U-test [21].

First, we analyse the efficacy of the employed TCG algorithms and how they compare against one another. We will then evaluate the automated TCG algorithms against the manually created test suite.

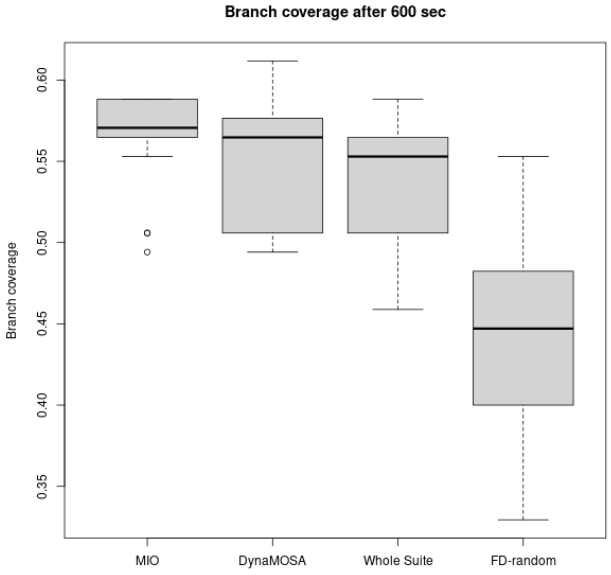
5.1 Comparing the TCG algorithm with each other

Figure 5 shows the final distributions of branch and mutation coverages for all four automated TCG algorithms after 600s. When we compare the final coverages in Figure 5 of the different automated TCG algorithms, we see that all three state-of-the-art approaches improve upon the FD-random in terms of branch coverage. This is further confirmed by the statistical tests – MIO reaches significantly higher branch coverage than:

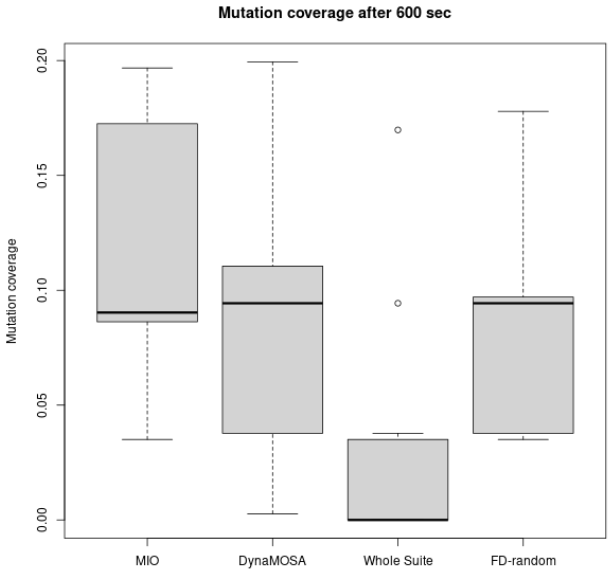
- DynaMOSA ($p = 0.007396$),
- Whole Suite Generation ($p = 1.019 \cdot 10^{-6}$), and
- FD-random ($p = 2.2 \cdot 10^{-16}$).

While the differences between DynaMOSA and Whole Suite Generation are not significant, our data shows that DynaMOSA is significantly better than FD-random generation ($p = 2.464 \cdot 10^{-16}$). Similarly, Whole Suite Generation outperforms FD-random generation with a p value of $1.489 \cdot 10^{-15}$. These results confirm the finding in the literature. We can also observe and confirm the faster increase in coverage in DynaMOSA as presented by Panichella et al. [27].

Looking at the mutation coverage metric in Figure 5, however, we notice an overall low sensitivity of the generated test suites. While we observe a significantly higher mutation coverage when we compare MIO to Whole Suite Generation ($p = 6.653 \cdot 10^{-16}$) and DynaMOSA to Whole Suite Generation ($p = 4.002 \cdot 10^{-14}$), the Whole Suite Generation is significantly worse than FD-random generation ($p = 4.17 \cdot 10^{-14}$). The remaining pairings show no statistically significant difference. The problem in sensitivity is also apparent when comparing the generated test suites and their reached mutation coverages to the original baseline test suite. The data for which is provided in Table 1. Here, only MIO ($p = 0.04647$)



(a) Final branch coverages per algorithm



(b) Final mutation coverages per algorithm

Figure 5: Coverages across all algorithms after the timeout of 600s. We use the same boxplot type as described earlier for Figure 3.

and FD-random generation ($p = 0.04699$) show a statistically significant increase in sensitivity. Both DynaMOSA and Whole Suite Generation fail to do so. When looking at the raw aggregated data in Table 1, we see that even the maximum increase in mutation coverage over all four algorithms only constitutes an improvement of approximately 0.16.

5.2 Comparing the TCG algorithms with the manually created test suite

Comparing the coverages reached by the automated TCG algorithms with our manually created test suite shows a dire picture: Not a single approach even comes close. The manually created test suite reaches the maximum possible value in the branch coverage and approximately 90.5% mutation coverage. The manually created test suite is significantly better in both coverage metrics than any of the automatically generated test suites. Table 2 shows the p values for each of the comparisons. In all cases, the null hypothesis was rejected. This is not surprising – even if we look at the results most favourable for the automated approaches in Table 1, we see differences of at least 0.388 for branch coverage and of at least 0.71 for mutation coverage.

5.3 Discussion

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%. This value is too low to assure the software's quality [12]. We argue that a reason for the low coverage and sensitivity of the test suites may be system states that are difficult to reach. For example: the `exciting` Calculator contains a *read* method. In order to run the method fully, a correct output file of the Fortran `exciting` DFT code is required. To construct such a file, the test cases generation would have needed to create a conforming output file or test case that manages to run the entire DFT code successfully. Thus, fully testing the *read* method is unlikely for the used TCG tools. This improbability of randomly generating conforming complex structured input for certain methods under test appears to be a general problem for the state-of-the-art TCG algorithms. Recently, Olsthoorn et al.[23] presented an approach integrating grammar-based fuzzing with TCG, partly solving the issue of complex structured input generation.

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 certain methods.

6 THREATS TO VALIDITY AND FUTURE WORK

All employed TCG algorithms in this case study rely on randomness to generate any given test case in some way. Therefore, it is possible that especially useful or particularly problematic decisions were taken during the generation of the test suites. To mitigate the uncertainty stemming from the randomness in the algorithms, we ran each algorithm 50 times. We provide publicly for independent review, all configuration files, seed values, and scripts for all experimental runs.[7]

[7] https://doi.org/10.5281/zenodo.6364844

Null hypothesis	Alternate hypothesis	p value Branch cov.	p value Mutation cov.
MIO > Manual	MIO ≤ Manual	**0.03833**	**0.04647**
DynaMOSA > Manual	DynaMOSA ≤ Manual	**0.04466**	**0.04742**
Whole Suite > Manual	Whole Suite ≤ Manual	**0.04575**	**0.03241**
FD-random > Manual	FD-random ≤ Manual	**0.04736**	**0.04699**

Table 2: Comparison of the coverage metrics of the automatically generated test suites and the manually created test suite. A p value lower than 0.05 indicates rejection of the null hypothesis.

Secondly, the employed software may contain errors itself. We tried to mitigate this threat by using standardized software packages wherever possible: This includes the TCG algorithms, the calculation of coverage metrics, as well as the statistical evaluation scripts. We did not change the TCG configurations and used the default values provided by Pynguin. This means, however, that these defaults may not be the best settings for our use-case. We will leave the evaluation of parameter tuning for a future study. In a similar vein, Cosmic-Ray may not have detected all semantically equivalent mutants. Thus, in a future study, we should evaluate whether Cosmic-Ray indeed does find all equivalent mutants.

We noted that both DynaMOSA and Whole Suite Generation led to several program crashes of Pynguin. We could not discover the root cause for this, however. Therefore, we dealt with these internal tool crashes by restarting the crashed experimental runs. Preliminary data from our current research seems to indicate that custom exceptions may be able to reproduce this behavior.

Finally, the employed case study may not be generalizable. While these results could potentially hold true for other ASE *Calculators*, this likely is not the case for all types of Python projects or even other scientific software. We are currently investigating this question in another study with Elastic 2.0 – another material science project. Preliminary data seems to confirm our findings presented in this study. Further confirmation studies in other fields may be needed, which we leave for future work.

Due to time constraints, we also leave for future work the investigation whether creating an ensemble test suite out of all automatically generated test cases would achieve higher coverage and mutation score than the manually-written one. Furthermore, it is unclear why MIO outperforms DynaMOSA in this particular use-case – we also leave a study investigating the cause for this behavior for future work.

7 CONCLUSION

In our case study, we compared the (i) branch, and (ii) mutation coverage reached after 600 seconds across the four different TCG algorithms that Pynguin supports. We also compare the results to those of a manually created test suite.

We observed on average a 0.30 increase in branch coverage over the baseline of the original test suite. Using the automated TCG algorithms that metric reached up to 55%. In terms of mutation coverage, we saw an average increase of 0.07 over the baseline for an average value of 10%. These values are statistically significant improvements.

However, when we compare the manually created gold standard to the baseline test suite, we see an increase in branch coverage of 75% to 100%; as well as an increase in mutation coverage of 87% to 90%. Consequently, the manually created test suite also achieves much higher coverage metrics when compared to the test suites that were automatically generated through the four TCG algorithms.

Interpreting the results of this case study, we argue that 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. Furthermore, we reaffirm the need to investigate how test case generation algorithms may generate test cases where variables are meaningfully named. Test cases that follow such a naming scheme would allow testers to use the contextual clues from the variable names to better locate faults from automatically generated test cases.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 414984028 – SFB 1404 FONDA.

REFERENCES

[1] Andrea Arcuri. 2017. Many Independent Objective (MIO) Algorithm for Test Suite Generation. In *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings (Lecture Notes in Computer Science)*, Tim Menzies and Justyna Petke (Eds.), Vol. 10452. Springer, 3–17. https://doi.org/10.1007/978-3-319-66299-2_1

[2] Ermira Daka. 2018. *Improving readability in automatic unit test generation.* Ph.D. Dissertation. University of Sheffield, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.755187

[3] Ermira Daka, José Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. 2015. Generating Readable Unit Tests for Guava. In *Search-Based Software Engineering - 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Márcio de Oliveira Barros and Yvan Labiche (Eds.), Vol. 9275. Springer, 235–241. https://doi.org/10.1007/978-3-319-22183-0_17

[4] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 107–118. https://doi.org/10.1145/2786805.2786838

[5] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating unit tests with descriptive names or: would you name your children thing1 and thing2?. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 57–67. https://doi.org/10.1145/3092703.3092727

[6] Matheus Ferreira, Lincoln Costa, and Francisco Carlos Souza. 2020. Search-based Test Data Generation for Mutation Testing: a tool for Python programs. In *Anais da IV Escola Regional de Engenharia de Software*. SBC, 116–125.

[7] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *19th Symposium on the Foundations of Software Engineering (FSE) and 13th European Software Engineering Conference (ESEC)*. ACM, 416–419.

[8] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Software Eng.* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[9] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 8:1–8:42.

[10] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering* 20, 3 (2015), 611–639.

[11] Andris Gulans, Stefan Kontur, Christian Meisenbichler, Dmitrii Nabok, Pasquale Pavone, Santiago Rigamonti, Stephan Sagmeister, Ute Werner, and Claudia Draxl. 2014. Exciting: a full-potential all-electron package implementing density-functional theory and many-body perturbation theory. *Journal of Physics: Condensed Matter* 26, 36 (2014), 363202.

[12] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*. IEEE, 30–39.

[13] Pierre Hohenberg and Walter Kohn. 1964. Inhomogeneous electron gas. *Physical review* 136, 3B (1964), B864.

[14] Daniel Hook and Diane Kelly. 2009. Testing for trustworthiness in scientific software. In *ICSE Workshop on Software Engineering for Computational Science and Engineering, SE-CSE 2009, Vancouver, BC, Canada, May 23, 2009*. IEEE Computer Society, 59–64. https://doi.org/10.1109/SECSE.2009.5069163

[15] Diane Kelly and Rebecca Sanders. 2008. The challenge of testing scientific software. In *Proceedings of the 3rd annual conference of the Association for Software Testing (CAST 2008: Beyond the Boundaries)*. Citeseer, 30–36.

[16] D. Kelly, S. Thorsteinson, and D. Hook. 2011. Scientific Software Testing: Analysis with Four Dimensions. *IEEE Softw.* 28, 3 (2011), 84–90. https://doi.org/10.1109/MS.2010.88

[17] 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. 2017. The atomic simulation environment—a Python library for working with atoms. *Journal of Physics: Condensed Matter* 29, 27 (2017), 273002.

[18] Mingyi Lim, Giovani Guizzo, and Justyna Petke. 2020. Impact of Test Suite Coverage on Overfitting in Genetic Improvement of Software. In *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings (Lecture Notes in Computer Science)*, Aldeida Aleti and Annibale Panichella (Eds.), Vol. 12420. Springer, 188–203. https://doi.org/10.1007/978-3-030-59762-7_14

[19] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings (Lecture Notes in Computer Science)*, Aldeida Aleti and Annibale Panichella (Eds.), Vol. 12420. Springer, 9–24. https://doi.org/10.1007/978-3-030-59762-7_2

[20] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Automated Unit Test Generation for Python. *CoRR* abs/2111.05003 (2021). arXiv:2111.05003 https://arxiv.org/abs/2111.05003

[21] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[22] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.

[23] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1224–1228.

[24] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *Future of Software Engineering Proceedings (FOSE 2014)*. ACM, 117–132.

[25] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 815–816. https://doi.org/10.1145/1297846.1297902

[26] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 75–84. https://doi.org/10.1109/ICSE.2007.37

[27] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Trans. Software Eng.* 44, 2 (2018), 122–158.

[28] Paolo Tonella. 2004. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA*, George S. Avrunin and Gregg Rothermel (Eds.). ACM, 119–128.

[29] Kevin Tran, Aini Palizhati, Seoin Back, and Zachary W Ulissi. 2018. Dynamic workflows for routine materials discovery in surface science. *Journal of chemical information and modeling* 58, 12 (2018), 2392–2400.

[30] Thomas Vogel, Stephan Druskat, Markus Scheidgen, Claudia Draxl, and Lars Grunske. 2019. Challenges for verifying and validating scientific software in computational materials science. In *Proceedings of the 14th International Workshop on Software Engineering for Science, SE4Science@ICSE 2019, Montreal, QC, Canada, May 28, 2019*. IEEE / ACM, 25–32. https://doi.org/10.1109/SE4Science.2019.00010