

Pynguin: Automated Unit Test Generation for Python

Stephan Lukasczyk

stephan.lukasczyk@uni-passau.de
University of Passau
Passau, Germany

Gordon Fraser

gordon.fraser@uni-passau.de
University of Passau
Passau, Germany

ABSTRACT

Automated unit test generation is a well-known methodology aiming to reduce the developers' effort of writing tests manually. Prior research focused mainly on statically typed programming languages like Java. In practice, however, dynamically typed languages have received a huge gain in popularity over the last decade. This introduces the need for tools and research on test generation for these languages, too. We introduce PYNGUIN, an extendable test-generation framework for Python, which generates regression tests with high code coverage. PYNGUIN is designed to be easily usable by practitioners; it is also extensible to allow researchers to adapt it for their needs and to enable future research. We provide a demo of PYNGUIN at <https://youtu.be/UiGrG25Vts0>; further information, documentation, the tool, and its source code are available at <https://www.pynguin.eu>.

CCS CONCEPTS

• **Software and its engineering** → *Search-based software engineering*; **Software testing and debugging**; *Software maintenance tools*.

KEYWORDS

Python, Automated Test Generation

ACM Reference Format:

Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion)*, May 21–29, 2022, Pittsburgh, PA, USA. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3510454.3516829>

1 INTRODUCTION

Automated software test generation has a long history both in research and industrial settings. Over the years, researchers have presented many approaches for generating test input data, such as random [4] and search-based [14] techniques. Many of these approaches have been implemented in tools, most notably RAN-DOOP [15] and EvoSUITE [5] for the Java programming language.

The focus on the Java programming language, however, represents a limitation of prior research, as dynamically typed programming languages such as JavaScript and Python have gained

huge popularity over the last decade. Python, in particular, is popular in many domains such as data science and machine learning, and it nowadays ranks as one of the most used programming languages (see, for example, the IEEE Spectrum ranking¹). This increasing popularity of the language requires more and better tools to support the developers and improve the general code quality they produce. Testing is one of the most important techniques to improve the quality of software, but automated tool support for test generation is currently lacking in the Python tool box. Unfortunately, automated test generation is challenging in the context of a dynamically typed language.

A crucial problem impeding the development of test generation techniques is the fact that programs written in a dynamically typed language usually do not provide any information about variable types. Statically deriving this information is hard [8], as these languages often allow to change the type of a variable's value throughout the program, dynamically modify objects at runtime, or provide type coercions that might not match the intent of the programmer [19]. As a consequence, test generation research has so far only tackled specific aspects or restricted scenarios. For example, TSTL [9], CROSSHAIR², KLARA³, or AUGER⁴, require manual effort by the developer before they can produce test cases. Although HYPOTHESIS [13] can also generate tests automatically, its aim is to find minimal property-violating input values. We therefore introduce PYNGUIN [10], which aims to use search-based test-generation techniques to automatically generate regression tests [22] with high code coverage, without requiring user input.

PYNGUIN is an open-source framework written in and for the Python programming language. It uses search-based test generation to generate tests that maximise code coverage. PYNGUIN by default incorporates type information into the test-generation process. However, it is also able to generate covering test cases for programs that do not explicitly provide type information. Designed as an extendable framework, PYNGUIN allows both researchers and practitioners to explore established and new ideas for test generation in the context of a dynamically typed programming language, such as further coverage criteria or new test-generation algorithms.

Initial empirical evaluation [10] of PYNGUIN shows that the automated generation of regression tests is feasible also for a dynamically typed programming language: on average, PYNGUIN achieved a branch coverage of up to 68.0 % on 118 Python modules from 17 open-source libraries. Our initial evaluation furthermore indicates that type information is crucial also for the test-generation process: Incorporating type information leads to significantly higher coverage levels; our previous experiments show a median improvement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9223-5/22/05...\$15.00
<https://doi.org/10.1145/3510454.3516829>

¹<https://spectrum.ieee.org/top-programming-languages/>, last accessed 2022-02-10.

²<https://crosshair.readthedocs.io>, last accessed 2022-02-10.

³<https://klara-py.readthedocs.io>, last accessed 2022-02-10.

⁴<https://github.com/laffra/auger>, last accessed 2022-02-10.

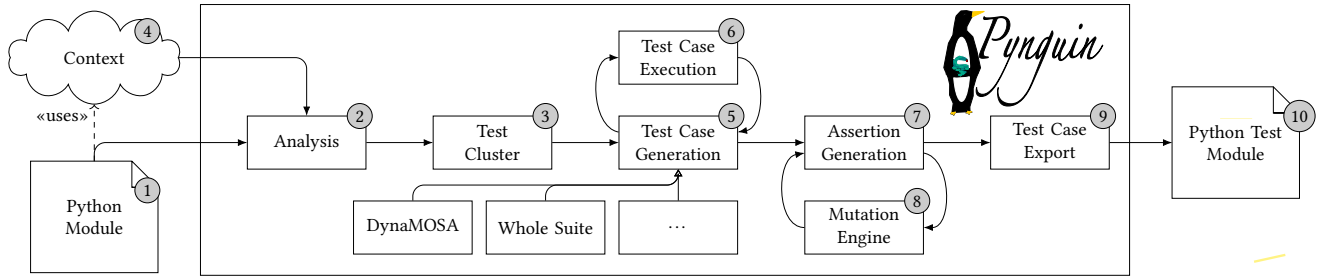


Figure 1: The execution steps of PYNGUIN.

of up to 2.7 percentage points over all used projects, depending on the test-generation algorithm [11]. This paper describes the inner workings of PYNGUIN and how one can use and extend it.

2 TEST GENERATION WITH PYNGUIN

PYNGUIN is written in Python and requires at least Python 3.8 to run. It can, however, generate unit tests also for Python projects that are built for older versions of Python. PYNGUIN can be run as a standalone command-line application or – which is recommended – inside a DOCKER container. It is released under the GNU LGPL open-source licence.

2.1 PYNGUIN’s Components

Figure 1 shows the components of PYNGUIN and their interactions throughout the test-generation process.

PYNGUIN takes as input a Python module (denoted by ① in Fig. 1).⁵ It then analyses the module to extract information ②. The extracted information consists, among others, of the declared classes, functions, and methods. From this information PYNGUIN builds the so-called *test cluster* [21] ③. The test cluster contains all information about the module under test, most importantly, which classes, functions, and methods are declared, and what their parameters are. Furthermore, PYNGUIN inspects the modules that are transitively included by the module under test (shown as the *context* in Fig. 1, ④). From the context, PYNGUIN extracts the types they define by searching for those class definitions that are available in the namespace of the module under test. These types are then used as input-type candidates during the test-generation phase. PYNGUIN selects classes, methods, and functions from the test cluster during the generation to build the test cases.

When constructing a test case, PYNGUIN selects a function or method from the module under test. Consider the example code snippet in Listing 1: there is only one function in the module, `triangle`, which PYNGUIN selects as its target function. It therefore adds a statement representing a method call to `triangle` to its internal test-case representation. Afterwards, PYNGUIN aims to fulfill the requirements of the function’s parameters in a backwards fashion. In the example, PYNGUIN knows from the type annotations that `int` statements are required. It therefore generates one to three variable assignment statements of the form `var = <num>` and adds them to the test case before the function-call statement. The number of `int`

statements as well as the generated values are chosen randomly by PYNGUIN, because variable values can be used for more than one parameter. Listing 2 shows two test cases that have been created by this way. In case a more complex object is required as a parameter, PYNGUIN will attempt to generate it by recursively fulfilling the parameters of the involved methods; the necessary statements are also prepended to the list of statements of the test case. We provide a detailed example of this process in our previous work [11].

For test input generation ⑤ the user can select between various well-established algorithms: [DynaMOSA \[17\]](#), [MIO \[1\]](#), [MOSA \[16\]](#), [random \[15\]](#), [Whole Suite \[6\]](#), and [Whole Suite with archive \[20\]](#). Depending on the selected algorithm, PYNGUIN generates one or many test cases. It then executes the newly generated test cases against the module under test to measure the achieved coverage ⑥. Currently, PYNGUIN can consider line or branch coverage as an optimisation goal for its search algorithms. To support other variants of coverage one needs to provide further fitness functions; further coverage criteria are planned for future work. It is possible to select one sort of coverage for the optimisation or a combination of many. To measure coverage we instrument Python’s byte code on-the-fly to trace which parts of the module under test have been executed by a generated test. After evaluating fitness, PYNGUIN continues with the next iteration of the test-generation algorithm. This process stops once a configurable stopping condition is satisfied, such as a time limit or a predefined amount of algorithm iterations. It is also possible to stop the generation after all coverage goals have been met, which means the generated tests achieve 100 % coverage.

After the test-input generation PYNGUIN optionally attempts to generate regression assertions [22] to not only execute the code under test but also check its results ⑦. The approach implemented in PYNGUIN is based on mutation testing [7]. PYNGUIN utilises a customised version of `MUTPY` [3] to generate mutated versions from the original module under test ⑧. `MUTPY` executes the tests generated by the previous stage of PYNGUIN against these mutants as well as the original module. By tracing the values of object attributes and function returns, PYNGUIN determines which values change on the mutated version, compared to the original module. For these values PYNGUIN generates assertions that interpret the returned values on the original module as the ground truth. As a consequence, the generated assertions are able to kill the aforementioned mutants if they show different behaviour compared to the original module. An advantage of generating regression tests this way is that it implicitly minimises the number of assertions present in the resulting test cases.

⁵Usually, a module in Python is equivalent to a source file. We currently restrict the support to modules written in Python because of the necessary code instrumentation.

Finally, PYNGUIN generates Python source code ⑨ from its internal representation of the test cases and exports the source code in the style of the popular PyTest⁶ framework into a Python module ⑩. Further styles, for example, unittest from Python's standard API, can also be integrated easily.

Each of the stages of PYNGUIN is built as modular and as independent of the others as possible. PYNGUIN itself is furthermore built with extensibility in mind. This allows to replace stages and components easily.

2.2 Using PYNGUIN

PYNGUIN is written in Python. It can thus be either executed after checking out its source code or — more conveniently — be installed from the Python Package Index (PyPI)⁷ via the pip utility tool.

The primary usage of PYNGUIN is as a command-line application. It also provides a rudimentary API that allows controlling the framework from inside another application without the need to launch an external process. As future work we plan to enhance PYNGUIN's public API such that it can also be used as a library for test generation within other projects without the necessity to execute it as a standalone application. Our presentation here, however, will only discuss the command-line interface. We refer the interested reader to PYNGUIN's documentation⁸, which describes the API. One can get an overview of all command-line arguments using the --help option after installing PYNGUIN:

```
$ pynguin --help
```

Please note that PYNGUIN requires the user to set the environment variable PYNGUIN_DANGER_AWARE; PYNGUIN executes the code under test with arbitrary random inputs. Depending on the code under test this can cause side effects and harm to the user's system. By setting the environment variable to an arbitrary value the user confirms to PYNGUIN that they are aware of this risk.

The main arguments of PYNGUIN are --project-path to specify the path of the project PYNGUIN should generate tests for, --module-name to specify the name of the module to generate tests for, and the --output-path, where PYNGUIN stores the generated test cases. PYNGUIN requires the user to set at least those three parameters; all further arguments come with documented default values yielded by the --help parameter.

Consider the example in Listing 1, saved to a module triangle.py in the current work directory. One can now run PYNGUIN with minimal configuration options:

```
$ pynguin \
  --project-path ./ \
  --output-path /tmp/pynguin-tests \
  --module-name triangle
```

This results in PYNGUIN generating test cases using DynaMOSA (the current default algorithm). It stores them in files to the folder /tmp/pynguin-tests. Now suppose that the user wants to generate tests for the same module but with the MIO algorithm instead of DynaMOSA. All they need to do is to add --algorithm MIO to their command line:

```
def triangle(x: int, y: int, z: int) -> str:
    if x == y == z:
        return "Equilateral triangle"
    elif x == y or y == z or x == z:
        return "Isosceles triangle"
    else:
        return "Scalene triangle"
```

Listing 1: A simple function checking for triangle properties.

```
import example as module0

def test_case_0():
    int_0 = 4271
    int_1 = -3706
    str_0 = module0.triangle(int_0, int_0, int_1)
    assert str_0 == "Isosceles triangle"

def test_case_1():
    int_0 = -163
    int_1 = 484
    int_2 = 77
    str_0 = module0.triangle(int_0, int_1, int_2)
    assert str_0 == "Scalene triangle"
```

Listing 2: An excerpt of the test cases generated by PYNGUIN.

```
$ pynguin \
  --project-path ./ \
  --output-path /tmp/pynguin-results \
  --module-name triangle \
  --algorithm MIO
```

Similarly, one can set further configuration options. Listing 2 shows an excerpt of the generated result.⁹ The two shown test cases in the result execute the triangle function with different parameter values to execute different branches of the function's implementation; the test cases also provide assertions that check on the returned value of the triangle function. Please note that re-executing PYNGUIN will overwrite the resulting files.

In the aforementioned settings, PYNGUIN will not print any output to the terminal. A more verbose output can be achieved by adding the -v or -vv parameter.

2.3 Dynamic Typing

As Python is a dynamically typed language, it does not require the user to specify any type information, although recent versions of the language support annotations for such information. PYNGUIN aims to parse type annotations from the source code if they are available (configurable as a parameter to PYNGUIN). Any parsed information about parameter types of functions and methods as well about their return types is incorporated into the test cluster. The type information stored in the test cluster allows PYNGUIN to select specific objects to satisfy the requirements of the parameters when generating test cases for a specific function or method.

Besides extracting type information from annotations in the source code, PYNGUIN can also be extended to query external type-inference tools. If no type annotations are available for the code under test PYNGUIN considers all available types from the test cluster as

⁶<https://pytest.org>, last access 2022-02-10.

⁷<https://pypi.org/project/pynguin>, last accessed 2022-02-10.

⁸<https://pynguin.readthedocs.io>, last accessed 2022-02-10.

⁹Since the generation process is based on random numbers, the exact result might differ. It still should look similar.

candidates during input generation. In this case, PYNGUIN currently selects one of the available types from the test cluster randomly.

Let us again consider the triangle-classification function from Listing 1. Suppose there were no type annotation present: As a consequence, PYNGUIN could only guess the types of the parameters and might come up with objects of arbitrary types available from the test cluster. Since our example does neither define new types nor import any modules, only the so called ‘builtins’ are available: basic types such as `float` or `str`. Choosing parameter values of type `float` instead of `int` would be a reasonable choice for this triangle function. However, the following test case would also be valid for the Python language if there is no type information available:

```
import example as module0

def test_case_2():
    list_0 = ["foo", "bar"]
    str_0 = module0.triangle(list_0, list_0, list_0)
    assert str_0 == "Equilateral triangle"
```

Checking the triangle type for a list of strings is not a reasonable thing, although the language would permit it. Note that the above test case is valid because it executes the program without crashes and covers parts of the code of the `triangle` function; it therefore contributes towards the optimisation goal of high coverage. However, the test case also shows that coverage may not be a good metric for the effects of types. Using an unexpected type as an input may often also simply lead to crashes of the program under test, for example, if the code attempts to access non-existing attributes. Furthermore, the lack of type information can also prevent PYNGUIN from being able to instantiate the correct objects. This simple example shows that type information is crucial for test generation to generate not only covering but also valid and useful tests. While we strongly advocate the usage of type annotations to improve PYNGUIN’s resulting test cases, the use of alternative means such as type inference is an open research problem. Enabling future research to address this problem is a core motivation for building PYNGUIN.

2.4 Extending PYNGUIN

PYNGUIN is a framework that allows easy extension at various points. Many extensions can be built by implementing only few classes. For example, adding a new test-generation algorithm can be achieved by extending the abstract class `TestGenerationStrategy` and implementing its `generate_tests` method. It is also necessary to register the new algorithm in the configuration and the algorithm instantiation. The algorithm instantiation allows providing predefined operators to the new algorithm such as a factory for new chromosomes or fitness functions for the search objectives. We designed these components with evolutionary algorithms in mind; one can of course also define purely random-based algorithms as we have done with our Random algorithm, based on the RANDOOP algorithm [15].

To demonstrate how simple this can be, Listing 3 implements an algorithm that is based on random test-case sampling: it generates a random test case by adding a method call and its dependencies — the object to call the method on as well as objects to fill the parameter values (see Sec. 2 of our previous work [11] for a detailed illustration of this process). Our example algorithm utilises the archive that is

```
class RandomSamplingStrategy(TestGenerationStrategy):
    def generate_tests(self):
        self.before_search_start()
        solution = self._chromosome_factory.get_chromosome()
        self._archive.update([solution])
        test_suite = self.create_test_suite(
            self._archive.solutions
        )
        self.before_first_search_iteration(solution)

        while (
            self.resources_left()
            and test_suite.get_fitness() != 0.0
        ):
            new = self._chromosome_factory.get_chromosome()
            self._archive.update([new])
            test_suite = self.create_test_suite(
                self._archive.solutions
            )
            self.after_search_iteration(test_suite)

        self.after_search_finish()
        return self.create_test_suite(
            self._archive.solutions
        )
```

Listing 3: A random sampling strategy for test input generation.

also used for the MOSA [16] algorithm to store generated test cases that cover certain coverage goals in order to keep track of them. Now the algorithm loops until the stopping condition is fulfilled or the test cases stored in the archive cover all coverage goals. In each loop iteration, the algorithm randomly samples another test case. It stores the new test case in the archive as well. The given implementation also calls some helper methods that keep track of the test-generation process; for example, they track the achieved coverage value over the generation time. In the end, the example algorithm creates a test suite, that is, a collection of the generated test cases, and returns them back to the framework. We already implemented this algorithm in PYNGUIN; one can select it by setting `--algorithm RANDOM_TEST_CASE_SEARCH`.

Similarly, other parts of PYNGUIN can be extended. Examples are test-case export in different styles by implementing an AST visitor or the incorporation of type-inference techniques by querying external type-inference tools.

3 EVALUATION

We evaluated PYNGUIN by conducting a small experiment for this paper, using the aforementioned algorithms for test generation. We used 118 modules from our previous work [11] for our evaluation and ran PYNGUIN in version 0.17.0 [12] 30 times on each module and in each configuration to minimise the influence of randomness. For the experiment, we set the timeout for test generation to 600 s. In this work we only give few insights; for a more extensive evaluation we refer the reader to our previous work [10, 11], which not only studies the differences between various algorithms in greater detail but also investigates on the influence of type information.

To gain insights on the performance of the different algorithms, we measured branch coverage. Figure 2 shows the development of the mean coverage per configuration over the generation time of

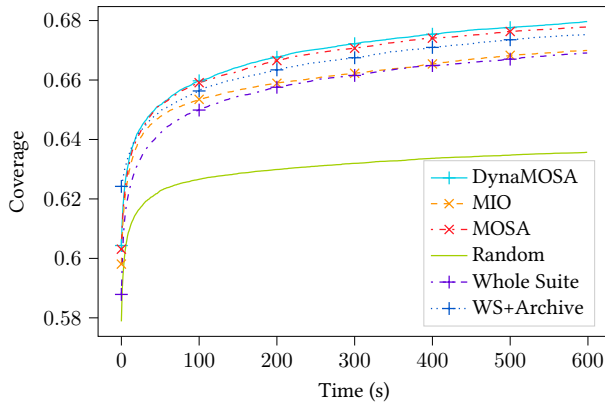


Figure 2: Development of the coverage over time.

600 s. One can clearly see that the search-based techniques outperform the random algorithm. The five search-based algorithms, however, only show small differences, with DynaMOSA achieving the highest and Whole Suite the lowest coverage values (mean branch coverage for DynaMOSA: 68.0 %, MIO: 67.0 %, MOSA: 67.8 %, Random: 63.6 %, Whole Suite: 66.9 %, and for Whole Suite with archive: 67.5 %). These results are in line with previous research [2, 18] in the context of statically typed languages: the search-based algorithms achieve a higher average coverage than the random algorithm. Furthermore, DynaMOSA yields the highest coverage values.

4 CONCLUSIONS

The increasing popularity of Python requires the availability of many types of tools to aid the developers. We introduce PYNGUIN, an automated unit test generation framework, designed to support developers when implementing unit tests manually. PYNGUIN is available as a command-line application, which is the de-facto standard for many developer-aiding tools. While this provides great flexibility for users, PYNGUIN's modular design also enables further extensions and research in the field of automated unit test generation for dynamically typed languages.

In this work we have summarised the features of PYNGUIN. By providing PYNGUIN as open source, we hope to foster further research as well as its applicability in practice. Further information on PYNGUIN, its documentation, and source code are available at

<https://www.pynguin.eu>

ACKNOWLEDGMENTS

This work is supported by DFG project FR 2955/4-1.

REFERENCES

- [1] Andrea Arcuri. 2017. Many Independent Objective (MIO) Algorithm for Test Suite Generation. In *International Symposium on Search Based Software Engineering (SSBSE) (Lecture Notes in Computer Science, Vol. 10452)*. Springer, 3–17. https://doi.org/10.1007/978-3-319-66299-2_1
- [2] José Campos, Yan Ge, Nasser Albulian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information & Software Technology* 104 (2018), 207–235. <https://doi.org/10.1016/j.infsof.2018.08.010>
- [3] Anna Derezinska and Konrad Halas. 2014. Experimental Evaluation of Mutation Testing Approaches to Python Programs. In *International Conference on Software Testing, Verification and Validation Workshops (ICST-Workshops)*. IEEE Computer Society, 156–164. <https://doi.org/10.1109/ICSTW.2014.24>
- [4] Joe W. Duran and Simeon C. Ntafos. 1984. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering* 10, 4 (1984), 438–444. <https://doi.org/10.1109/TSE.1984.5010257>
- [5] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 416–419. <https://doi.org/10.1145/2025113.2025179>
- [6] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. <https://doi.org/10.1109/TSE.2012.14>
- [7] Gordon Fraser and Andreas Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering* 38, 2 (2012), 278–292. <https://doi.org/10.1109/TSE.2011.93>
- [8] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 94–105. <https://doi.org/10.1145/2771783.2771809>
- [9] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, and James O'Brien. 2018. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer* 20, 1 (2018), 57–78. <https://doi.org/10.1007/s10009-016-0445-y>
- [10] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *International Symposium on Search Based Software Engineering (SSBSE) (Lecture Notes in Computer Science, Vol. 12420)*. Springer, 9–24. https://doi.org/10.1007/978-3-030-59762-7_2
- [11] 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
- [12] Stephan Lukasczyk, Florian Kroiß, Gordon Fraser, and Pynguin Contributors. 2022. *se2p/pynguin: Pynguin v0.17.0*. <https://doi.org/10.5281/zenodo.5971799>
- [13] David MacIver and Zac Hatfield-Dodds. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. <https://doi.org/10.21105/joss.01891>
- [14] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Journal of Software Testing, Verification and Reliability* 14, 2 (2004), 105–156. <https://doi.org/10.1002/stvr.294>
- [15] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 75–84. <https://doi.org/10.1109/ICSE.2007.37>
- [16] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 1–10. <https://doi.org/10.1109/ICST.2015.7102604>
- [17] 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 Transactions on Software Engineering* 44, 2 (2018), 122–158. <https://doi.org/10.1109/TSE.2017.2663435>
- [18] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information & Software Technology* 104 (2018), 236–256. <https://doi.org/10.1016/j.infsof.2018.08.009>
- [19] Michael Pradel and Koushik Sen. 2015. The Good, the Bad, and the Ugly: An Empirical Study on Implicit Type Conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 519–541. <https://doi.org/10.4230/LIPIcs.ECOOP.2015.519>
- [20] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893. <https://doi.org/10.1007/s10664-015-9424-2>
- [21] Stefan Wappler and Frank Lammermann. 2005. Using evolutionary algorithms for the unit testing of object-oriented software. In *Annual Conference on Genetic and Evolutionary Computation (GECCO)*. 1053–1060. <https://doi.org/10.1145/1068009.1068187>
- [22] Tao Xie. 2006. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 4067)*. Springer, 380–403. https://doi.org/10.1007/11785477_23