Empirical Software Engineering manuscript No.
(will be inserted by the editor)

An Empirical Study of Automated Unit Test Generation for Python

Stephan Lukasczyk · Florian Kroiß · Gordon Fraser

Received: date / Accepted: date

Abstract Various mature automated test generation tools exist for statically typed programming languages such as Java. Automatically generating unit tests for dynamically typed programming languages such as Python, however, is substantially more difficult due to the dynamic nature of these languages as well as the lack of type information. Our Pynguin framework provides automated unit test generation for Python. In this paper, we extend our previous work on Pynguin to support more aspects of the Python language, and by studying a larger variety of well-established state of the art test-generation algorithms, namely DynaMOSA, MIO, and MOSA. Furthermore, we improved our Pynguin tool to generate regression assertions, whose quality we also evaluate. Our experiments confirm that evolutionary algorithms can outperform random test generation also in the context of Python, and similar to the Java world, DynaMOSA yields the highest coverage results. However, our results also demonstrate that there are still fundamental remaining issues, such as inferring type information for code without this information, currently limiting the effectiveness of test generation for Python.

Keywords Dynamic Typing · Python · Automated Test Generation

S. Lukasczyk
University of Passau
Innstr. 33, 94032 Passau, Germany
E-mail: stephan.lukasczyk@uni-passau.de

F. Kroiß
University of Passau
Innstr. 33, 94032 Passau, Germany
E-mail: kroiss@fim.uni-passau.de

G. Fraser
University of Passau
Innstr. 33, 94032 Passau, Germany
E-mail: gordon.fraser@uni-passau.de

1 Introduction

Automated unit test generation is an established field in research and a technique well-received by researchers and practitioners to support programmers. Mature research prototypes exist, implementing test-generation approaches such as feedback-directed random generation (Pacheco et al. 2007) or evolutionary algorithms (Campos et al. 2018). These techniques enable the automated generation of unit tests for statically typed programming languages, such as Java, and remove the burden of the potentially tedious task of writing unit tests from the programmer.

In recent years, however, dynamically typed programming languages, most notably JavaScript, Python, and Ruby, have gained huge popularity amongst practitioners. Current rankings, such as the IEEE Spectrum Ranking[1] underline this popularity, now listing Python as the overall most popular language according to their ranking criteria. Dynamically typed languages lack static type information on purpose as this is supposed to enable rapid development (Gong et al. 2015). However, this dynamic nature has also been reported to cause reduced development productivity (Kleinschmager et al. 2012), code usability (Mayer et al. 2012), or code quality (Meyerovich and Rabkin 2013; Gao et al. 2017). The lack of a (visible static) type system is the main reason for type errors encountered in these dynamic languages (Xu et al. 2016).

The lack of static types is particularly problematic for automated test generation, which requires type information to provide appropriate parameter types for method calls or to assemble complex objects. In absence of type information the test generation can only guess, for example, the appropriate parameter types for new function calls. To overcome this limitation, existing test generators for dynamically typed languages often do not target test-generation for general APIs, but resort to other means such as using the document object model of a web browser to generate tests for JavaScript (Mirshokraie et al. 2015), or by targeting specific systems, such as the browser's event handling system (Artzi et al. 2011; Li et al. 2014). A general purpose unit test generator at the API level has not been available until recently.

Pynguin (Lukasczyk et al. 2020; Lukasczyk and Fraser 2022) aims to fill this fundamental gap: Pynguin is an automated unit test generation framework for Python programs. It takes a Python module as input together with the module's dependencies, and aims to automatically generate unit tests that maximise code coverage. The version of Pynguin we evaluated in our previous work (Lukasczyk et al. 2020) implemented two established test generation techniques: whole-suite generation (Fraser and Arcuri 2013) and feedback-directed random generation (Pacheco et al. 2007). Our empirical evaluation showed that the whole-suite approach is able to achieve higher code coverage than random testing. Furthermore, we studied the influence of available type

[1] `https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020`, accessed 2021–02–02.

information, which leads to higher resulting coverage if it can be utilised by the test generator.

In this paper we extend our previous evaluation (Lukasczyk et al. 2020) in several aspects and make the following contributions: We implemented further test generation algorithms: the many-objective sorting algorithm (MOSA) (Panichella et al. 2015) and its extension DynaMOSA (Panichella et al. 2018a), as well as the many independent objective (MIO) algorithm (Arcuri 2017, 2018). We study and compare the performance of all algorithms in terms of resulting branch coverage. We furthermore enlarge the corpus of subjects for test generation to get a broader insight into test generation for Python. For this we take projects from the BugsInPy project (Widyasari et al. 2020) and the ManyTypes4Py dataset (Mir et al. 2021); in total, we add eleven new projects to our dataset. Additionally, we implemented the generation of regression assertions based on mutation testing (Fraser and Zeller 2012).

Our empirical evaluation confirms that, similar to prior findings on Java, DynaMOSA and MOSA perform best on Python code, with a median of 80.7 % branch coverage. Furthermore, our results show that type information is an important contributor to coverage on all studied test-generation algorithms, although this effect is only significant for some of our subject systems, showing an average impact of 2.20 % to 4.30 % on coverage. While this confirms that type information is one of the challenges in testing Python code, these results also suggest there are other open challenges for future research, such as dealing with type hierarchies and constructing complex objects.

2 Background

The main approaches to automatically generate unit tests are either by creating random sequences, or by applying meta-heuristic search algorithms. Random testing assembles sequences of calls to constructors and methods randomly, often with the objective to find undeclared exceptions (Csallner and Smaragdakis 2004) or violations of general object contracts (Pacheco et al. 2007), but the generated tests can also be used as automated regression tests. The effectiveness of random test generators can be increased by integrating heuristics (Ma et al. 2015; Sakti et al. 2015). Search-based approaches use a similar representation, but apply evolutionary search algorithms to maximize code coverage (Tonella 2004; Baresi and Miraz 2010; Fraser and Arcuri 2013; Andrews et al. 2011).

As an example to illustrate how type information is used by existing test generators, consider the following snippets of Java (left) and Python (right) code:

```
class Foo {
  Foo(Bar b) { ... }
  void doFoo(Bar b) { ... } }
class Bar {
  Bar() { ... }
  Bar doBar(Bar b) { ... } }
```

```
class Foo:
  def __init__(self, b): ...
  def do_foo(self, b): ...
class Bar:
  def __init__(self): ...
  def do_bar(self, b): ...
```

Assume `Foo` of the Java example is the class under test. It has a dependency on class `Bar`: in order to generate an object of type `Foo` we need an instance of `Bar`, and the method `doFoo` also requires a parameter of type `Bar`.

Random test generation would typically generate tests in a forward way. Starting with an empty sequence $t_0 = \langle\rangle$, all available calls for which all parameters can be satisfied with objects already existing in the sequence can be selected. In our example, initially only the constructor of `Bar` can be called, since all other methods and constructors require a parameter, resulting in $t_1 = \langle o_1 = \texttt{new Bar()}\rangle$. Since t_1 contains an object of type `Bar`, in the second step the test generator now has a choice of either invoking `doBar` on that object (and use the same object also as parameter), or invoking the constructor of `Foo`. Assuming the chosen call is the constructor of `Foo`, we now have $t_2 = \langle o_1 = \texttt{new Bar()}; o_2 = \texttt{new Foo}(o_1);\rangle$. Since there now is also an instance of `Foo` in the sequence, in the next step also the method `doFoo` is an option. The random test generator will continue extending the sequence in this manner, possibly integrating heuristics to select more relevant calls, or to decide when to start with a new sequence.

An alternative approach, for example applied during the mutation step of an evolutionary test generator, is to select necessary calls in a backwards fashion. That is, a search-based test generator like EvoSuite (Fraser and Arcuri 2013) would first decide that it needs to, for example, call method `doFoo` of class `Foo`. In order to achieve this, it requires an instance of `Foo` and an instance of `Bar` to satisfy the dependencies. To generate a parameter object of type `Bar`, the test generator would consider all calls that are declared to return an instance of `Bar`—which is the case for the constructor of `Bar` in our example, so it would prepend a call to `Bar()` before the invocation of `doFoo`. Furthermore, it would try to instantiate `Foo` by calling the constructor. This, in turn, requires an instance of `Bar`, for which the test generator might use the existing instance, or could invoke the constructor of `Bar`.

In both scenarios, type information is crucial: In the forward construction type information is used to inform the choice of call to append to the sequence, while in the backward construction type information is used to select generators of dependency objects. Without type information, which is the case with the Python example, a forward construction (1) has to allow all possible functions at all steps, thus may not only select the constructor of `Bar`, but also that of `Foo` with an arbitrary parameter type, and (2) has to consider all existing objects for all parameters of a selected call, and thus, for example, also `str` or `int`. Backwards construction without type information would also have to try to select generators from all possible calls, and all possible objects, which both result in a potentially large search space to select from.

Without type information in our example we might see instantiations of `Foo` or calls to `do_foo` with parameter values of unexpected types, such as:

```
var_1 = Bar()
var_2 = Foo(42)
var_3 = var_2.do_foo("hello")
```

This example violates the assumptions of the programmer, which are that the constructor of `Foo` and the `do_foo` method both expect an object of type `Bar`. When type information is not present such test cases can be generated and will only fail if the execution raises an exception; for example, due to a call to a method that is defined for the `Bar` type but not on an `int` or `str` object. Type information can be provided in two ways in recent Python versions: either in a stub file that contains type hints or directly annotated in the source code. A stub file can be compared to C header files: they contain, for example, method declarations with their according types. Since Python 3.5, the types can also be annotated directly in the implementing source code, in a similar fashion known from statically typed languages (see PEP 484[2]).

3 Unit Test Generation with Pynguin

We introduce our automated test-generation framework for Python, called Pynguin, in the following sections. We start with a general overview on Pynguin. Afterwards, we formally introduce a representation for the test-generation problem using evolutionary algorithms in Python. We also discuss different components and operators that we use.

3.1 The Pynguin Framework

Pynguin is a framework for automated unit test generation written in and for the Python programming language. The framework is available as open-source software licensed under the GNU Lesser General Public License from its GitHub repository[3]. It can also be installed from the Python Package Index (PyPI)[4] using the `pip` utility. We refer the reader to Pynguin's web site[5] for further links and information on the tool.

Pynguin takes as input a Python module and allows the generation of unit tests using different techniques. For this, it analyses the module and extracts information about available methods from the module and types from the module and its transitive dependencies to build a test cluster (see Section 3.3). Next, it generates test cases using a variety of different test-generation algorithms (which we describe in the following sub-sections). Afterwards, it generates regression assertions for the previously generated test cases (see Section 3.7). Finally, a tool run emits the generated test cases in the style of the widely-used PyTest[6] framework. Figure 1 illustrates Pynguin's components and their relationships. For a detailed description of the components, how to use Pynguin for own purposes, as well as how to extend Pynguin, we refer the reader to our respective tool paper (Lukasczyk and Fraser 2022).

[2] `https://python.org/dev/peps/pep-0484/`, accessed 2021–05–04.
[3] `https://github.com/se2p/pynguin`, accessed 2022–07–06.
[4] `https://pypi.org/project/pynguin/`, accessed 2022–07–06.
[5] `https://www.pynguin.eu`, accessed 2022–07–06.
[6] `https://www.pytest.org`, accessed 2022–07–06.

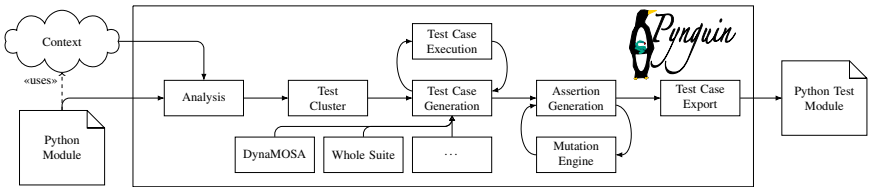

Fig. 1: The different components of PYNGUIN (taken from Lukasczyk and Fraser (2022))

Algorithm 1: Feedback-directed random generation (adapted from Pacheco et al. (2007))

Input: Stopping condition C, a set of modules c
Output: A pair of non-error sequences s_n and error sequences s_e

$s_e \leftarrow \{\}$;
$s_n \leftarrow \{\}$;
while $\neg C$ **do**
  $m(T_1 \dots T_k) \leftarrow$ RANDOMPUBLICMETHOD(c);
  $\langle S, V \rangle \leftarrow$ RANDOMSEQSANDVALS($s_n, T_1 \dots T_k$;
  $N \leftarrow$ EXTEND(m, S, V);
  if $N \in s_n \cup s_e$ **then**
    Continue
  $\langle \vec{\mathbf{o}}, v \rangle \leftarrow$ EXECUTE(N);
  if v **then**
    $s_e \leftarrow s_e \cup \{N\}$;
  else
    $s_n \leftarrow s_n \cup \{N\}$;
return $\langle s_n, s_e \rangle$

PYNGUIN is built to be extensible with other test generation approaches and algorithms. It comes with a variety of well-received test-generation algorithms, which we describe in the following sections.

3.1.1 Feedback-Directed Random Test Generation

PYNGUIN provides a feedback-directed random algorithm adopted from RANDOOP (Pacheco et al. 2007). The algorithm starts with two empty test suites, one for passing and one for failing tests, and randomly adds statements to an empty test case. The test case is executed after each addition. Test cases that do not raise an exception are added to the passing test suite; otherwise they are added to the failing test suite. The algorithm will then randomly choose a test case from the passing test suite or an empty test case as the basis to add further statements. Algorithm 1 shows the basic algorithm.

The main differences of our implementation compared to RANDOOP are that our version of the algorithm does not check for contract violations, does not implement any of RANDOOP's filtering criteria, and it does not require the user to provide a list of relevant classes, functions, and methods. Please

Algorithm 2: Monotonic Genetic Algorithm (adopted from Campos et al. (2018))

Input: Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

$P \leftarrow$ GenerateRandomPopulation(p_s);
PerformFitnessEvaluation(δ, P);
while $\neg C$ **do**
  $N_P \leftarrow \{\} \cup$ Elitism(P);
  while $|N_P| < p_s$ **do**
    $p_1, p_2 \leftarrow$ Selection(s_f, P);
    $o_1, o_2 \leftarrow$ Crossover(c_f, c_p, p_1, p_2);
    Mutation(m_f, m_p, o_1);
    Mutation(m_f, m_p, o_2);
    PerformFitnessEvaluation(δ, o_1);
    PerformFitnessEvaluation(δ, o_2);
    if Best(o_1, o_2) *is better than* Best(p_1, p_2) **then**
      $N_P \leftarrow N_P \cup \{o_1, o_2\}$;
    else
      $N_P \leftarrow N_P \cup \{p_1, p_2\}$;
  $P \leftarrow N_P$;
return P

note that our random algorithm, in contrast to the following evolutionary algorithms, does not use a fitness function to guide its random search process. It only checks the coverage of the generated test suite to stop early once the module under test is fully covered. We will refer to this algorithm as *Random* in the following.

3.1.2 Whole Suite Test Generation

The whole suite approach implements a genetic algorithm that takes a test suite, that is a set of test cases, as an individual (Fraser and Arcuri 2013). It uses a monotonic genetic algorithm, as shown in Algorithm 2 as its basis. Mutation and crossover operators can modify the test suite as well as its contents, that is, the test cases. It uses the sum of all branch distances as a fitness function. We will refer to this algorithm as *WS* in the following.

3.1.3 Many-Objective Sorting Algorithm (MOSA)

The *Many-Objective Sorting Algorithm* (MOSA, (Panichella et al. 2015)) is an evolutionary algorithm specifically designed to foster branch coverage and overcome some of the limitations of whole-suite generation. MOSA considers branch coverage as a many-objective optimisation problem, whereas previous approaches, such as whole suite test generation, combined the objectives into one single value. It therefore assigns each branch its individual objective function. The basic MOSA algorithm is shown in Algorithm 3.

Algorithm 3: Many-Objective Sorting Algorithm (MOSA, adopted from Campos et al. (2018))

Input: Stopping condition C, Fitness function δ, Population size p_s, Crossover function c_f, Crossover probability c_p, Mutation probability m_p
Output: Archive of optimised individuals A

$p \leftarrow 0$;
$N_p \leftarrow$ GenerateRandomPopulation(p_s);
PerformFitnessEvaluation(δ, N_p);
$A \leftarrow \{\}$;
while $\neg C$ **do**
    $N_o \leftarrow$ GenerateOffspring(c_f, c_p, m_p, N_p);
    $R_t \leftarrow N_p \cup N_o$;
    $r \leftarrow 0$;
    $F_r \leftarrow$ PreferenceSorting(R_t);
    $N_{p+1} \leftarrow \{\}$;
    while $|N_{p+1}| + |F_r| \leq p_s$ **do**
        CalculateCrowdingDistance(F_r);
        $N_{p+1} \leftarrow N_{p+1} \cup F_r$;
        $r \leftarrow r + 1$;
    DistanceCrowdingSort(F_r);
    $N_{p+1} \leftarrow N_{p+1} \cup F_r$ with size $p_s - |N_{p+1}|$;
    UpdateArchive(A, N_{p+1});
    $p \leftarrow p + 1$;
return A

MOSA starts with an initial random population and evolves this population to improve its ability to cover more branches. Earlier many-objective genetic algorithms suffer from the so-called dominance resistance problem (von Lücken et al. 2014). This means that the proportion of non-dominated solutions increases exponentially with the number of goals to optimise. As a consequence the search process degrades to a random one. MOSA specifically targets this by introducing a preference criterion to choose the optimisation targets to focus the search only on the still relevant targets, that is, the yet uncovered branches. Furthermore, MOSA keeps a second population, called *archive*, to store already found solutions, that is, test cases that cover branches. This archive is built in a way that it automatically prefers shorter test cases over longer for the same coverage goals. We refer the reader to the literature for details on MOSA (Panichella et al. 2015). In the following we will refer to this algorithm as *MOSA*.

3.1.4 Dynamic Target Selection MOSA (DynaMOSA)

DynaMOSA (Panichella et al. 2018a) is an extension to the original MOSA algorithm. The novel contribution of DynaMOSA was to dynamically select the targets for the optimisation. This selection is done on the control-dependency hierarchy of statements. Let us consider the Python code snippet from Listing 1.

```
1 if foo < 42:
2     if bar == 23:
3         do_something()
```

Listing 1: A Python snippet showing control-dependent conditions.

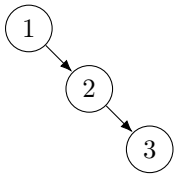

Fig. 2: The control-dependence graph for the snippet in Listing 1.

The condition in line 2 is control dependent on the condition in the first line. This means, it can only be reached and covered, if the condition in the first line is satisfied. Thus, searching for an assignment for the variable `bar` to fulfil the condition—and thus covering line 3—is not necessary unless the condition in line 1 is fulfilled. Figure 2 depicts the control-dependence graph. DynaMOSA uses the control-dependencies, which are found in the control-dependence graph of the respective code, to determine which goals to select for further optimisation. To compute the control dependencies within Pynguin, we generate a control-flow graph from the module under test's byte code using the `bytecode`[7] library; we use standard algorithms to compute post-dominator tree and control-dependence graph from the control-flow graph (see, for example, the work of Ferrante et al. (1987) for such algorithms). We refer the reader to the literature for details on DynaMOSA (Panichella et al. 2018a). In the following we will refer to this algorithm as *DynaMOSA*.

3.1.5 Many Independent Objectives (MIO)

The *Many Independent Objectives* Algorithm (MIO, (Arcuri 2017)) targets some limitations of both the whole suite and the MOSA approach. To do this, it combines the simplicity of a $(1 + 1)$EA with feedback-directed target selection, dynamic exploration/exploitation, and a dynamic population. MIO also maintains an archive of tests, were it keeps a different population of tests for each testing target, for example, for each branch to cover. Algorithm 4 shows its main algorithm.

MIO was designed with the aim of overcoming some intrinsic limitations of the Whole Suite or MOSA algorithms that arise especially in system-level test generation. Such systems can contain hundreds of thousands of objectives to cover, for which a fixed-size population will most likely be not suitable; hence, MIO uses a dynamic population. It furthermore turns out that exploration is good at the beginning of the search whereas a focused exploitation is beneficial for better results in the end; MIO addresses this insight by a dynamic exploration/exploitation control. Lastly, again addressing the large number of objectives and limited resources, MIO selects the objectives to focus its search on by using a feedback-directed sampling technique. The literature (Arcuri

[7] `https://www.pypi.org/project/bytecode`, accessed 2022–07–14.

Algorithm 4: Many Independent Objective (MIO) Algorithm (adapted from Campos et al. (2018))

Input: Stopping condition C, Fitness function δ, Population size N, Number of mutations M, Mutation function m_f, Mutation probability m_p, Probability of random sampling R, Start of focus search F
Output: Archive of optimised individuals A

```
1  Z ← SetOfEmptyPopulations();
2  A ← {};
3  p ← null;
4  m ← 1;
5  while ¬C do
6      if p ≠ null ∧ m < M then
7          p ← Mutation(m_f, m_p, p);
8          m ← m + 1;
9      else if p = null ∨ R > RANDOM(0, 1) then
10         p ← GenerateRandomIndividual();
11         m ← 1;
12     else
13         p ← SampleIndividual(Z);
14         p ← Mutation(m_f, m_p, p);
15         m ← 1;
16     forall t ∈ ReachedTargets(p) do
17         if IsTargetCovered(t) then
18             UpdateArchive(A, p);
19             Z ← Z \ {Z_t};
20         else
21             Z_t ← Z ∪ {p};
22             if |Z_t| > N then
23                 RemoveWorstTest(Z_t, δ);
24     UpdateParameters(F, R, N, M);
25 return A
```

2017, 2018) provides more details on MIO to the interested reader. We will refer to this algorithm as *MIO* in the following.

3.2 Problem Representation

As the *unit* for unit test generation, we consider Python *modules*. A module is usually identical with a file and contains definitions of, for example, functions, classes, or statements; these can be nested almost arbitrarily. When the module is loaded the definitions and statements at the top level are executed. While generating tests we do not only want all definitions to be executed, but also all structures defined by those definitions, for example, functions, closures, or list comprehensions. Thus, in order to apply a search algorithm, we first need to define a proper representation of the valid solutions for this problem.

We use a representation based on prior work from the domain of testing Java code (Fraser and Arcuri 2013). For each statement s_j in a test case t_i we assign one value $v(s_j)$ with type $\tau(v(s_j)) \in \mathcal{T}$, with the finite set of types $\mathcal{T}$ used in the subject-under-test (SUT) and the modules transitively imported by the

SUT. A set of test cases form a *test suite*. We define five kinds of statements: *Primitive statements* represent `int`, `float`, `bool`, `bytes` and `str` variables, for example, `var_0 = 42`. Value and type of a statement are defined by the primitive variable. Note that although in Python everything is an object, we treat these values as primitives because they do not require further construction in Python's syntax.

Constructor statements create new instances of a class, for example, `var_0 = SomeType()`. Value and type are defined by the constructed object; any parameters are satisfied from the set $V = \{v(s_k) \mid 0 \leq k < j\}$. *Method statements* invoke methods on objects, for example, `var_1 = var_0.foo()`. Value and type are defined by the return value of the method; source object and any parameters are satisfied from the set V. *Function statements* invoke functions, for example, `var_2 = bar()`. They do not require a source object but are otherwise identical to method statements.

Extending our previous work (Lukasczyk et al. 2020) we introduce the generation of collection statements. *Collection statements* create new collections, that is, `List`, `Set`, `Dict`, and `Tuple`. An example for such a list collection statement is `var_2 = [var_0, var_1]`; an example for a dictionary collection statement is `var_4 = {var_0: var_1, var_2: var_3}`. Value and type are defined by the constructed collection; elements of a collection are satisfied from the set V. For dictionaries, both keys and values are satisfied from V. Tuples are treated similar to lists; their sole difference in Python is that lists are mutable while tuples are immutable.

Previously, we always filled in all parameters (except `*args` and `**kwargs`), when creating a constructor, method or function statement and passed the parameters by position. However, filling all parameters might not be necessary, as some parameters may have default values or are optional (for example `*args` and `**kwargs`, which will result in an empty tuple or dictionary, respectively). It can also be impossible to pass certain parameters by position as it is possible to restrict them to be only passed by *keyword*. We improved our representation of statements with parameters, by (1) passing parameters in the correct way, that is, *positional* or by *keyword*, and (2) leaving optional parameters empty with some probability.

Parameters of the form `*args` or `**kwargs` capture positional or keyword arguments which are not bound to any other parameter. Hereby, `args` and `kwargs` are just names for the formal parameters; they can be chosen arbitrarily, but `args` and `kwargs` are the most common names. Their values can be accessed as a tuple (`*args`) or a dictionary (`**kwargs`). We fill these parameters by constructing a list or dictionary of appropriate type and passing its elements as arguments by using the `*` or `**` unpacking operator, respectively. Consider the example snippet in Listing 2 to shed light into how a test case for functions involving those parameter types may look like.

This representation is of variable size; we constrain the size of test cases $l \in [1, L]$ and test suites $n \in [1, N]$. In contrast to prior work on testing Java (Fraser and Arcuri 2013), we do not define attribute or assignment statements; attributes of objects are not explicitly declared in Python but assigned dynamically,

Listing 2: Example test cases for functions accepting lists or dictionaries

```
def my_sum(*args):
    result = 0
    for x in args:
        result += x
    return result

# Test case for my_sum
def test_case_0():
    int_0 = 42
    list_0 = [int_0, int_0]
    # Equivalent to my_sum(int_0, int_0)
    int_1 = my_sum(*list_0)
    assert int_1 == 84

def concatenate(**kwargs):
    result = ""
    for key, value in kwargs.items():
        result += key + "=" + value + ";"
    return result

# Test case for concatenate
def test_case_1():
    str_0 = 'foo'
    str_1 = 'bar'
    dict_0 = {str_0: str_1}
    # Equivalent to concatenate(foo=str_1)
    str_2 = concatenate(**dict_0)
    assert str_2 == 'foo=bar;'
```

hence it is non-trivial to identify the existing attributes of an object and we leave it as future work. Assignment statements in the Java representation could assign values to array indices. This was necessary, as Java arrays can only be manipulated using the `[]`-operator. While Python also has a `[]`-operator, the same effect can also be achieved by directly calling the `__setitem__` or `__getitem__` methods. Please note that we do not use the latter approach in Pynguin currently, because Pynguin considers all methods having names starting with one or more underscores to be private methods; private methods are not part of the public interface of a module and thus Pynguin does not directly call them. Given these constraints, we currently cannot generate a test case as depicted in Listing 3 because this would require some of the aforementioned features, such as reading and writing attributes or the `[]`-operator for list manipulation. The latter is currently not implemented in Pynguin because we assume that changing the value of a list element is not often required; it is more important to append values to lists, which is supported by Pynguin. Additionally, in Java an array has a fixed size, whereas lists in Python have variable size. This would require a way to find out a valid list index that could be used for the `[]`-operator.

Listing 3: A test case with statements that are currently not supported

```
def test_case_0():
    obj_0 = SomeClass()
    # Attribute read not supported
    var_0 = obj_0.foo

    int_0 = 42
    # Attribute write not supported
    obj_0.bar = int_0

    int_1 = 0
    int_2 = 23
    list_0 = [int_0]
    # []-operator not supported.
    list_0[int_1] = int_2
```

3.3 Test Cluster

For a given subject under test, the *test cluster* (Wappler and Lammermann 2005) defines the set of available functions and classes along with their methods and constructors. The generation of the test cluster recursively includes all imports from other modules, starting from the subject under test. The test cluster also includes all primitive types as well as Python's built-in collection types, that are, `List`, `Set`, `Dict`, and `Tuple`. To create the test cluster we load the module under test and inspect it using the `inspect` module from the standard Python API to retrieve all available functions and classes from this module. Additionally, we transitively inspect all dependent modules.

The resulting test cluster basically consists of two maps and one set: the set contains information about all callable or accessible elements in the module under test, which are classes, functions, and methods. This set also stores information about the fields of enums, as well as static fields at class or module level. During test generation, Pynguin selects from this set the callable or accessible elements in the module under test to generate its inputs for.

The two maps store information about which callable or accessible elements can generate a specific type, or modify it. Please note that these two maps do not only contain elements from the module under test but also from the dependent modules. Pynguin uses these two maps to generate or modify specific types, if needed.

3.4 Operators for the Genetic Algorithms

Except the presented random algorithm (see Section 3.1.1), the test-generation algorithms (see Sections 3.1.2 to 3.1.5) implemented in Pynguin are genetic algorithms. Genetic algorithms are inspired by natural evolution and have been used to address many different optimisation problems. They encode a solution to the problem as an individual, called *chromosome*; a set of individuals is

called *population*. Using operations inspired by genetics, the algorithm optimises the population gradually. Operations are, for example, *crossover*, *mutation*, and *selection*. Crossover merges genetic material from at least two individuals into a new offspring, while mutation independently changes the elements of an individual. Selection is being used to choose individuals for reproduction that are considered better with respect to some fitness criterion (Campos et al. 2018).

In the following, we introduce those operators and their implementation in PYNGUIN in detail.

3.4.1 The Crossover Operator

Crossover is used to merge the genetic material from at least two individuals into a new offspring. The different implemented genetic algorithms use different objects as their individuals: DynaMOSA, MIO, and MOSA consider a test case to be an individual, whereas Whole Suite considers a full test suite, consisting of potentially many test cases, to be an individual.

This makes it necessary to distinguish between *test-suite crossover* and *test-case crossover*; both work in a similar manner but have subtle differences.

Test-suite Crossover The search operators for our representation are based on those used in EVOSUITE (Fraser and Arcuri 2013): We use single-point relative crossover for both, crossing over test cases and test suites.

The crossover operator for *test suites*, which is used for the whole suite algorithm, takes two parent test suites P_1 and P_2 as input, and generates two offspring test suites O_1 and O_2, by splitting both P_1 and P_2 at the same relative location, exchanging the second parts and concatenating them with the first parts. Individual test cases have no dependencies between each other, thus the application of crossover always generates valid test suites as offspring. Furthermore, the operator decreases the difference in the number of test cases between the test suites, thus $\text{abs}(|O_1| - |O_2|) \leq \text{abs}(|P_1| - |P_2|)$. Therefore, no offspring will have more test cases than the larger of its parents.

Test-case Crossover For the implementation of DynaMOSA, MIO, and MOSA we also require a crossover operator for *test cases*. This operator works similar to the crossover operator for test suites. We describe the differences in the following using an example.

Listing 4 depicts an example defining a class `Foo` containing a constructor and two methods, as well as two test cases that construct instances of `Foo` and execute both methods. During crossover, each test case is divided into two parts by the crossover point and the latter parts of both test cases are exchanged, which may result in the test cases depicted in Listing 5. Since statements in the exchanged parts may depend on variables defined in the original first part, the statement or test case needs to be repaired to remain valid. For example, the insertion of the call `foo_0.baz(int_0)` into the first crossed-over test case requires an instance of `Foo` as well as an `int` value.

Listing 4: A class under test and two generated test cases before applying crossover

```
# Class under test
class Foo:
    def __init__(self, foo: str):
        self._foo = foo

    def bar(self, suffix: str) -> None:
        print(self._foo + suffix)

    def baz(self, repeat: int) -> None:
        print(self._foo * repeat)

# Test cases
def test_case_0():
    str_0 = "string a"
    foo_0 = Foo(str_0)
    str_1 = "string b"
    # <- Randomly chosen crossover point
    foo_0.bar(str_1)


def test_case_1():
    str_0 = "string c"
    foo_0 = Foo(str_0)
    int_0 = 1337
    # <- Randomly chosen crossover point
    foo_0.baz(int_0)
```

In the example, the crossover operator randomly decided to create a new `Foo` instance instead of reusing the existing `foo_0` as well as creating a new `int` constant to satisfy the `int` argument of the `baz` method. For the second crossed-over test case, the operator simply reused both, the instance of `Foo` as well the existing `str` constant to satisfy the requirements for the `bar` method.

3.4.2 The Mutation Operator

Similarly to the crossover operator, the different granularity of the individuals between the different genetic algorithms requires a different handling in the mutation operation.

Test-suite Mutation When mutating a *test suite* T, each of its test cases is mutated with probability $\frac{1}{|T|}$. After mutation, we add new randomly generated test cases to T. The first new test case is added with probability σ_{testcase}. If it is added, a second new test case is added with probability $\sigma^2_{\text{testcase}}$; this happens until the i-th test case is not added (probability: $1 - \sigma^i_{\text{testcase}}$). Test cases are only added if the limit N has not been reached, thus $|T| \leq N$.

Listing 5: Test cases from Listing 4 after performing crossover

```
def test_case_after_crossover_0():
    # Statements from test_case_0
    str_0 = "string a"
    foo_0 = Foo(str_0)  # Temporarily unused after crossover
    str_1 = "string b"  # Temporarily unused after crossover
    # Statement from test_case_1 + repairing statements
    foo_1 = Foo(str_0)
    int_0 = 42
    foo_1.baz(int_0)


def test_case_after_crossover_1():
    # Statements from test_case_1
    str_0 = "string c"
    foo_0 = Foo(str_0)
    int_0 = 1337        # Temporarily unused after crossover
    # Statement from test_case_0
    foo_0.bar(str_0)    # Used str_0 to satisfy parameter
```

Test-case Mutation The mutation of a *test case* can be one of three operations: *remove*, *change*, or *insert*, which we explain in the following sections. Each of these operations can happen with the same probability of $\frac{1}{3}$. A test case that has no statements left after the application of the mutation operator is removed from the test suite T.

When mutating a test case t whose last execution raised an exception at statement s_i, the following two rules apply in order:

1. If t has reached the maximum length, that is, $|t| \geq L$, the statements $\{s_j \mid i < j < l\}$ are removed from t.
2. Only the statements $\{s_j \mid 0 \leq j \leq i\}$ are considered for mutation, because the statements $\{s_j \mid i < j < l\}$ are never reached and thus have no impact on the execution result.

For constructing the initial population, a random test case t is sampled by uniformly choosing a value r with $1 \leq r \leq L$, and then applying the insertion operator repeatedly starting with an empty test case t', until $|t'| \geq r$.

The Insertion Mutation Operation With probability $\sigma_{\text{statement}}$ we insert a new statement at a random position $p \in [0, l]$. If it is inserted, we insert another statement with probability $\sigma^2_{\text{statement}}$ and so on, until the i-th statement is not inserted. New statements are only inserted, as long as the limit L has not been reached, that is, $|t| < L$.

For each insertion, with probability $\frac{1}{2}$ each, we either insert a new call on the module under test or we insert a method call on a value in the set $\{v(s_k) \mid 0 \leq k < p\}$. Any parameters of the selected call are either reused from the set $\{v(s_k) \mid 0 \leq k < p\}$, set to `None`, possibly left empty if they are optional (see Section 3.2), or are randomly generated. The type of a randomly

generated parameter is either defined by its type hint, or if not available, chosen randomly from the test cluster (see Section 3.3). If the type of the parameter is defined by a type hint, we can query the test cluster for callable elements in the subject under test or its dependencies that generate an object of the required type. Generic types currently cannot be handled properly in PYNGUIN, only Python's collection types are addressed. A parameter that has no type annotation or the annotation `Any`, requires us to consider all available types in the test cluster as potential candidates. For those, we can only randomly pick an element from the test cluster.

`*args`, `**kwargs` as well as parameters with a default value are only filled with a certain probability. For `*args: T` and `**kwargs: T` we try to create or reuse a parameter of type `List[T]` or `Dict[str, T]`, respectively. Primitive types are either randomly initialized within a certain range or reuse a value from static or dynamic constant seeding (Fraser and Arcuri 2012) with a certain probability. Complex types are constructed in a recursive backwards fashion, that is, by constructing their required parameters or reusing existing values.

The Change Mutation Operation For a test case $t = \langle s_0, s_1, \ldots, s_{l-1} \rangle$ of length l, each statement s_i is changed with probability $\frac{1}{l}$. For `int` and `float` primitives, we choose a random standard normally distributed value α. For `int` primitives we add $\alpha \Delta_{\text{int}}$ to the existing value. For `float` primitives we either add $\alpha \Delta_{\text{float}}$ or α to the existing value, or we change the amount of decimal digits of the current value to a random value in $[0, \Delta_{\text{digits}}]$. Here Δ_{int}, Δ_{float} and Δ_{digits} are constants.

For `str` primitives, with probability $\frac{1}{3}$ each, we delete, replace, and insert characters. Each character is deleted or replaced with probability $\frac{1}{|v(s_i)|}$. A new character is inserted at a random location with probability σ_{str}. If it is added, we add another character with probability σ_{str}^2 and so on, until the i-th character is not added. This is similar to how we add test cases to a test suite. `bytes` primitives are mutated similar to `str` primitives. For `bool` primitives, we simply negate $v(s_i)$.

For Tuples, we replace each of its elements with probability $\frac{1}{|v(s_i)|}$. Lists, sets, and dictionaries are mutated similar to how string primitives are mutated. Values for insertion or replacement are taken from $\{v(s_k) \mid 0 \leq k < i\}$. When mutating an entry of a dictionary, with probability $\frac{1}{2}$ we either replace the key or the value. For method, function, and constructor statements, we change each argument of a parameter with probability $\frac{1}{p}$, where p denotes the number of formal parameters of the callable used in s_i. For methods, this also includes the callee. If an argument is changed and the parameter is considered optional (see Section 3.2) then with a certain probability the associated argument is removed, if it was previously set, or set with a value from $\{v(s_k) \mid 0 \leq k < i\}$ if it was not set. Otherwise, we either replace the argument with a value from $\{v(s_k) \mid 0 \leq k < i\}$, whose type matches the type of the parameter or use `None`. If no argument was replaced, we replace the whole statement s_i by a call to another method, function, or constructor, which

is randomly chosen from the test cluster, has the same return type as $v(s_i)$, and whose parameters can be satisfied with values from $\{v(s_k) \mid 0 \leq k < i\}$.

The Remove Mutation Operation For a test case $t = \langle s_0, s_1, \ldots, s_{l-1} \rangle$ of length l, each statement s_i is deleted with probability $\frac{1}{l}$. As the value $v(s_i)$ might be used as a parameter in any of the statements $s_{i+1}, \ldots, s_{l-1}$, the test case needs to be repaired in order to remain valid. For each statement s_j, $i < j < l$, if s_j refers to $v(s_i)$, then this reference is replaced with another value out of the set $\{v(s_k) \mid 0 \leq k < j \wedge k \neq i\}$, which has the same type as $v(s_i)$. If this is not possible, then s_j is deleted as well recursively.

3.5 Covering and Tracing Python Code

A Python module contains various control structures, for example, `if` or `while` statements, which are guarded by logical predicates. The control structures are represented by conditional jumps at the bytecode level, based on either a unary or binary predicate. We focus on *branch coverage* in this work, which requires that each of those predicates evaluates to both true and false.

Let B denote the set of branches in the subject under test—two for each conditional jump in the byte code. Everything executable in Python is represented as a *code object*. For example, an entire module is represented as a code object, a function within that module is represented as another code object. We want to execute all code objects C of the subject under test. Therefore, we keep track of the executed code objects C_T as well as the minimum *branch distance* $d_{\min}(b, T)$ for each branch $b \in B$, when executing a test suite T. $B_T \subseteq B$ denotes the set of taken branches. Code objects which contain branches do not have to be considered as individual coverage targets, since covering one of their branches also covers the respective code object. Thus, we only consider the set of branch-less code objects $C_L \subseteq C$. We then define the branch coverage $\text{cov}(T)$ of a test suite T as $\text{cov}(T) = \frac{|C_T \cap C_L| + |B_T|}{|C_L| + |B|}$.

Branch distance is a heuristic to determine how far a predicate is away from evaluating to true or false, respectively. In contrast to previous work on Java, where most predicates at the bytecode level operate only on Boolean or numeric values, in our case the operands of a predicate can be any Python object. Thus, as noted by Arcuri (2013), we have to define our branch distance in such a way that it can handle arbitrary Python objects.

Let $\mathbb{O}$ be the set of possible Python objects and let $\Theta := \{\equiv, \not\equiv, <, \leq, >, \geq, \in, \notin, =, \neq\}$ be the set of binary comparison operators (remark: we use '$\equiv$', '$=$', and '$\in$' for Python's `==`, `is`, and `in` keywords, respectively). For each $\theta \in \Theta$, we define a function $\delta_\theta : \mathbb{O} \times \mathbb{O} \to \mathbb{R}_0^+ \cup \{\infty\}$ that computes the branch distance of the true branch of a predicate of the form $a\ \theta\ b$, with $a, b \in \mathbb{O}$ and $\theta \in \Theta$. By $\delta_{\bar{\theta}}(a, b)$ we denote the distance of the false branch, where $\bar{\theta}$ is the complementary operator of θ. Let further k be a positive number, and let $\text{lev}(x, y)$ denote the Levenshtein distance (Levenshtein 1966) between two strings x and y. The value of k is used in cases where we know that the distance

is not 0, but we cannot compute an actual distance value, for example, when a predicate compares two references for identity, the branch distance of the true branch is either 0, if the references point to the same object, or k, if they do not. While the actual value of k does not matter, we use $k = 1$.

The predicates is_numeric(z), is_string(z) and is_bytes(z) determine whether the type of their argument z is numeric, a string or a byte array, respectively. The function decode(z) decodes a byte array into a string by decoding every byte into a unique character, for example, by using the encoding ISO-8859-1.

$$\delta_{\equiv}(a,b) = \begin{cases} 0 & a \equiv b \\ |a-b| & a \not\equiv b \wedge \text{is\_numeric}(a) \wedge \text{is\_numeric}(b) \\ \text{lev}(a,b) & a \not\equiv b \wedge \text{is\_string}(a) \wedge \text{is\_string}(b) \\ \text{lev}(\text{decode}(a), \text{decode}(b)) & a \not\equiv b \wedge \text{is\_bytes}(a) \wedge \text{is\_bytes}(b) \\ \infty & \text{otherwise} \end{cases}$$

$$\delta_{<}(a,b) = \begin{cases} 0 & a < b \\ a - b + k & a \geq b \wedge \text{is\_numeric}(a) \wedge \text{is\_numeric}(b) \\ \infty & \text{otherwise} \end{cases}$$

$$\delta_{\leq}(a,b) = \begin{cases} 0 & a \leq b \\ a - b + k & a > b \wedge \text{is\_numeric}(a) \wedge \text{is\_numeric}(b) \\ \infty & \text{otherwise} \end{cases}$$

$$\delta_{>}(a,b) = \delta_{<}(b,a)$$

$$\delta_{\geq}(a,b) = \delta_{\leq}(b,a)$$

$$\delta_{\in}(a,b) = \begin{cases} 0 & a \in b \\ \min(\{\delta_{\equiv}(a,x) \mid x \in b\} \cup \{\infty\}) & \text{otherwise} \end{cases}$$

$$\delta_{\theta}(a,b) = \begin{cases} 0 & a\ \theta\ b \\ k & \text{otherwise} \end{cases} \qquad \theta \in \{\not\equiv, \notin, =, \neq\}$$

Note that every object in Python represents a Boolean value and can therefore be used as a predicate. Classes can define how their instances are coerced into a Boolean value, for example, numbers representing the value zero or empty collections are interpreted as false, whereas non-zero numbers or non-empty collections are interpreted as true. We assign a distance of k to the true branch, if such a unary predicate v represents false. Otherwise, we assign a distance of $\delta_F(v)$ to the false branch, where is_sized(z) is a predicate that determines if its argument z has a size and len(z) is a function that computes the size of its argument z.

$$\delta_F(a) = \begin{cases} \text{len}(a) & \text{is\_sized}(a) \\ |a| & \text{is\_numeric}(a) \\ \infty & \text{otherwise} \end{cases}$$

Future work shall refine the branch distance for different operators and operand types.

3.6 Fitness Functions

The fitness function required by genetic algorithms is an estimate of how close an individual is towards fulfilling a specific goal. As stated before we optimise our generated test suites towards maximum branch coverage. We define our fitness function with respect to this coverage criterion. Again, we need to distinguish between the fitness function for Whole Suite, which operates on the test-suite level, and the fitness function for DynaMOSA, MIO, and MOSA, which operates on the test-case level.

3.6.1 Test-suite Fitness

The fitness function required by our Whole Suite approach is constructed similar to the one used in EvoSuite (Fraser and Arcuri 2013) by incorporating the branch distance. The fitness function estimates how close a test suite is to covering *all* branches of the SUT. Thus, every predicate has to be executed at least twice, which we enforce in the same way as existing work (Fraser and Arcuri 2013): the actual branch distance $d(b, T)$ is given by

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice} \\ 1 & \text{otherwise} \end{cases}$$

with $\nu(x) = \frac{x}{x+1}$ being a normalisation function (Fraser and Arcuri 2013). Note that the requirement of a predicate being executed at least twice is used to avoid a circular behaviour during the search, where the whole suite algorithm alternates between covering a branch of a predicate, but in doing so loses coverage on the opposing branch (Fraser and Arcuri 2013).

Finally, we can define the resulting fitness function f of a test suite T as

$$f(T) = |C_L \setminus C_T| + \sum_{b \in B} d(b,T)$$

3.6.2 Test-case Fitness

The aforementioned fitness function is only applicable for the Whole Suite algorithm, which considers every coverage goal at the same time. DynaMOSA, MIO, and MOSA consider individual coverage goals. Such a goal is to cover either a certain branch-less code object $c \in C_L$ or a branch $b \in B$. For the former, its fitness is given by

$$f_c(t) = \begin{cases} 0 & c \text{ has been executed} \\ 1 & \text{otherwise} \end{cases}$$

while for the latter it is defined as

$$f_b(t) = al(b,t) + \nu(d_{min}(b,t))$$

where $d_{min}(b,t)$ is the smallest recorded branch distance of branch b when executing test case t and $al(b,t)$ denotes the approach level. The *approach level* is the number of control dependencies between the closest executed branch and b (Panichella et al. 2018a).

3.7 Assertion Generation

PYNGUIN aims to generate *regression* assertions (Xie 2006) within its generated test cases based on the values it observes during test-case execution. We consider the following types as *assertable*: enum values, builtin types (`int`, `str`, `bytes`, `bool`, and `None`), as well as builtin collection types (`tuple`, `list`, `dict`, and `set`) as long as they only consist of assertable elements. Additionally, PYNGUIN can generate assertions for `float` values using an approximate comparison to mitigate the issue of imprecise number representation. For those assertable types, PYNGUIN can generate the assertions directly, by creating an appropriate expected value for the assertion's comparison.

We call other types non-assertable. This does, however, not mean that PYNGUIN is not able to generate assertions for these types. It indicates that PYNGUIN will not try to create a specific object of such a type as an expected value for the assertion's comparison, but it will aim to check the object's state by inspecting its public attributes; assertions shall then be generated on the values of these attributes, if possible.

For non-assertable collections, PYNGUIN creates an assertion on their size by calling the `len` function on the collection and assert for that size. As a fallback, if none of the aforementioned strategies is successful, PYNGUIN is asserting that an object is not `None`. Listing 7 shows a simplified example of how test cases for the code snippet in Listing 6 could look like. Please note that the example was not directly generated with PYNGUIN but manually simplified for demonstration purpose.

Listing 6: A code snippet to generate tests with assertions for.

```
import queue

def divide(x: int, y: int) -> float:
  if y < 0:
    raise ValueError("y shall only be positive")
  return x / y

def queue():
  return queue.Queue()
```

Listing 7: Some test cases for the snippet in Listing 6.

```
import example as module0
import pytest

def test_case_0:
  int_0 = 1
  int_1 = 2
  float_0 = module0.divide(int_0, int_1)
  assert pytest.approx(0.5) == float_0

def test_case_1:
  var_0 = module_0.queue()
  assert var_0 is not None
```

Additionally, PYNGUIN creates assertions on raised exceptions. Python does, in contrast to Java, not know about the concept of checked and unchecked, or expected and unexpected exceptions. During PYNGUIN's analysis of the module under test we analyse the abstract syntax tree of the module and aim to extract for each function which exceptions it explicitly raises but not catches or that are specified in the function's Docstring. We consider these exceptions to be *expected*, all other exceptions that might appear during execution are *unexpected*. For an expected exception a generated test case will utilise the `pytest.raises` context from the PyTest framework[8], which checks whether an exception of a given type was raised during execution and causes a failure of the test case if not. PYNGUIN decorates a test case, who's execution causes an unexpected exception, with the `pytest.mark.xfail` decorator from the PyTest library; this decorator marks the test case as expected to be failing. Since PYNGUIN is not able to distinguish whether this failure is expected it is up to the user of PYNGUIN to inspect and deal with such a test case. Listing 8 shows two test cases that check for exceptions.

To generate assertions, PYNGUIN observes certain properties of the module under test. The parts of interest are the set of return values of function or method invocations and constructor calls, the static fields on used modules, and the static fields on seen return types. It does so after the execution of each statement of a test case. First, PYNGUIN executes every test case twice in random order. We do this to remove trivially flaky assertions, for example, assertions based on memory addresses or random numbers. Only assertions on values that are the same on both executions are kept as the *base assertions*.

To minimise the set of assertions (Fraser and Zeller 2012) to those that are likely to be relevant to ensure the quality of the current test case, PYNGUIN utilises a customised fork of MUTPY[9] to generate mutants of the module under test. A mutant is a variant of the original module under test that differs by few syntactic changes (Acree et al. 1978; DeMillo et al. 1978; Jia and Harman 2011).

[8] `https://www.pytest.org`, accessed 2022–07–13.

[9] `https://github.com/se2p/mutpy-pynguin`, accessed 2022–06–12.

Listing 8: Some test cases handling exceptions for the code snippet in Listing 6.

```
import example as module0
import pytest

def test_case_0:
  int_0 = 1
  int_1 = -1
  with pytest.raises(ValueError):
    module0.divide(int_0, int_1)

@pytest.mark.xfail(strict=True)
def test_case_0:
  int_0 = 1
  int_1 = 0
  module0.divide(int_0, int_1)
```

There exists a wide selection of so called *mutation operators* that generate mutants from a given program.

From the previous executions, PYNGUIN knows the result of each test case. It then executes all test cases against all mutants. If the outcome for one test case differs on a particular mutant between the original module and that mutant, the mutant is said to be *killed*; otherwise it is classified as *survived*. Out of the base assertions, only those assertions are considered relevant that have an impact on the result of the test execution. Such an impact means that the test is able to kill the mutant based on the provided assertions. Assertions that have not become relevant throughout all runs are removed from the test case, as they do not seem to help for fault finding.

3.8 Limitations of PYNGUIN

At its current state, PYNGUIN is still a research prototype facing several technical limitations and challenges. First, we noticed that PYNGUIN does not isolate the test executions properly in all cases. We used Docker containers to prevent PYNGUIN from causing harm on our systems; however, a proper isolation of file-system access or calls to specific functions, such as `sys.exit`, need to be handled differently. One of the reasons PYNGUIN fails on target modules is that they call `sys.exit` in their code, which stops the Python interpreter and thus also PYNGUIN. Future work needs to address those challenges by, for example, providing a virtualised file system to PYNGUIN executions or mocking calls to critical functions, such as `sys.exit`.

The dynamic nature of Python allows developers to change many properties of the program during runtime. One example is a non-uniform object layout; instances of the same class may have different methods or fields during runtime, of which we do not know when we analyse the class before execution. This is due to the fact that attributes can be added, removed, or changed during runtime, special methods like `__getattr__` can fake non-existing attributes

and many more. The work of Chen et al. (2018), for example, lists some of these dynamic features of the Python language and their impact on fixing bugs.

Besides the dynamic features of the programming language PYNGUIN suffers from supporting all language features. Constructs such as generators, iterators, decorators, coroutines, or higher-order functions provide open challenges for PYNGUIN and aim for future research. Furthermore, Python allows to implement modules in C/C++, which are then compiled to native binaries, in order to speed up execution. The native binaries, however, lack large parts of information that the Python source files provide, for example, there is no abstract syntax tree for such a module that could be analysed. PYNGUIN is not able to generate tests for a binary module because its coverage measurement relies on the available Python interpreter byte code that we instrument to measure coverage. Such an on-the-fly instrumentation is obviously not possible for binaries in native code.

Lastly, we want to note that the type system of Python is challenging. Consider the two types `List[Tuple[Any, ...]` and `List[Tuple[int, str]]`; there is no built-in support in the language to decide at runtime whether one is a subtype of the other. Huge engineering effort has been put into static type checkers, such as `mypy`[10], however they are not considered feature complete. Additionally, each new version of Python brings new features regarding type information to better support developers. This, however, also causes new challenges for tool developers to support those new features.

4 Empirical Evaluation

We use our PYNGUIN test-generation framework to empirically study automated unit test generation in Python. A crucial metric to measure the quality of a test suite is the coverage value it achieves; a test suite cannot reveal any faults in parts of the subject under test (SUT) that are not executed at all. Therefore, achieving high coverage is an essential property of a good test suite.

The characteristics of different test-generation approaches have been extensively studied in the context of statically typed languages, for example by Panichella et al. (2018b) or Campos et al. (2018). With this work we aim to determine whether these previous findings on the performance of test generation techniques also generalise to Python with respect to coverage:

Research Question 1 (RQ1) *How do the test-generation approaches compare on Python code?*

Our previous work (Lukasczyk et al. 2020) indicated that the availability of type information can be crucial to achieve higher coverage values. Type information, however, is not available for many programs written in dynamically typed languages. To gain information on the influence of type information on the different test-generation approaches, we extend our experiments from

[10] `http://mypy-lang.org/`, accessed 2022–07–14.

our previous study (Lukasczyk et al. 2020). Thus, we investigate the following research question:

Research Question 2 (RQ2) *How does the availability of type information influence test generation?*

Coverage is one central building block of a good test suite. Still, a test-generation tool solely focussing on coverage is not able to create a test suite that reveals many types of faults from the subject under test. The effectiveness of the generated assertions is therefore a crucial metric to reveal faults. We aim to investigate the quality of the generated assertions by asking:

Research Question 3 (RQ3) *How effective are the generated assertions at revealing faults?*

4.1 Experimental Setup

The following sections describe the setup that we used for our experiments.

4.1.1 Experiment Subjects

For our experiments we created a dataset of Python projects, which allows us to answer our research questions. We only selected projects for the dataset that contain type information in the style of PEP 484. Projects without type information would not allow us to investigate the influence of the type information (see RQ2). Additionally, we focus on projects that use features of the Python programming language that are supported by Pynguin. We do this to avoid skewing the results due to missing support for such features in Pynguin.

We excluded projects with dependencies to native code such as `numpy` from our dataset. We do this for two reasons: first, we want to avoid compiling these dependencies, which would require a C/C++ compilation environment in our experiment environment to keep this environment as small as possible. Second, Pynguin's module analysis and instrumentation rely on the source code and Python's internally used byte code. Both are not available for native code, which can limit Pynguin's applicability on such projects. Unfortunately, many projects depend to native-code libraries such as `numpy`, either directly or transitively. Pynguin can still be used on such projects but its capabilities might be limited by not being able to analyse these native-code dependencies. Fortunately, only few Python libraries are native-code only. Dependencies that only exist in native code, for example, the widely used `numpy` library, can still be used, although Pynguin might not be able to retrieve all information it could retrieve from a Python source file.

We reuse nine of the ten projects from our previous work (Lukasczyk et al. 2020); the removed project is `async_btree`, which provides an asynchronous behaviour-tree implementation. The functions in this project are implemented

as co-routines, which would require a proper handling of the induced parallelism and special function invocations. Pynguin currently only targets sequential code; it therefore cannot deal with this form of parallelism to avoid any unintended side effects. Thus, for the `async_btree` project, the test generation fails and only import coverage can be achieved, independent of the used configuration. None of the other used projects relies on co-routines.

We selected two projects from the BugsInPy dataset (Widyasari et al. 2020), a collection of 17 Python projects at the time of writing. The selected projects are those projects in the dataset that provide type annotations. The remaining nine projects have been randomly selected from the ManyTypes4Py dataset (Mir et al. 2021), a dataset of more than 5 200 Python repositories for evaluating machine learning-based type inference. We do not use all projects from ManyTypes4Py for various reasons: first, generating tests for 184 000 source code files from the dataset would cause a tremendous computational effort. Second, many of those projects also have dependencies on native-code libraries, which we explicitly excluded from our selection. Third, the selection criterion for projects in that dataset was that they use the MyPy type checker as a project dependency in some way. The authors state that only 27.6 % of the source-code files have type annotations. We manually inspected the projects from ManyTypes4Py we incorporated into our dataset to ensure that they have type annotations present.

Overall, this results in a set of 163 modules from 20 projects. Table 1 provides a more detailed overview of the projects. The column *Project Name* gives the name of the project on PyPI; the lines of code were measured using the cloc[11] tool. The table furthermore shows the total number of code objects and predicates in a project's modules, as well as the number of types detected per project. The former two measures give an insight on the project's complexity: higher numbers indicate larger complexity. The latter provides an overview how many types Pynguin was able to parse. Please note that Pynguin may not be able to resolve all types as it relies on Python's `inspect` library for this task. This library is the standard way of inspecting modules in Python—to extract information about existing classes and functions, or type information. If `inspect` fails to resolve a type that type will not be known to Pynguin's test cluster, which means that Pynguin will not try to generate an object of such a type. However, since the `inspect` library is part of the standard library we would consider its quality to be very good; furthermore, if such a type is reached in a transitive dependency, it might be used anyway.

4.1.2 Experiment Settings

We executed Pynguin on each of the constituent modules in sequence to generate test cases. We used Pynguin in version 0.25.2 (Lukasczyk et al. 2022) in a Docker container. The container is used for process isolation and is based

[11] `https://github.com/AlDanial/cloc`, accessed 2022–07–06.

Table 1: Projects used for evaluation

Project Name	Version	LOCs	Modules	CodeObjs.	Preds.	Types
`apimd`	1.2.1	390	1	50	110	10
`codetiming`	1.3.0	89	2	7	5	4
`dataclasses-json`	0.5.2	891	5	75	76	23
`docstring_parser`	0.7.3	506	4	50	86	9
`flake8`	3.9.0	542	8	80	41	0
`flutes`	0.3.0	235	3	4	0	8
`flutils`	0.7	772	6	48	136	19
`httpie`	2.4.0	1274	18	152	129	22
`isort`	5.8.0	841	6	37	9	11
`mimesis`	4.1.3	685	21	150	55	4
`pdir2`	0.3.2	461	5	43	46	9
`py-backwards`	0.7	618	18	125	100	13
`pyMonet`	0.12.0	394	8	145	59	6
`pypara`	0.0.24	877	7	232	70	15
`python-string-utils`	1.0.0	421	3	76	101	7
`pytutils`	0.4.1	943	19	62	80	4
`sanic`	21.3.2	1886	18	143	187	30
`sty`	1.0.0rc1	136	3	18	4	2
`thonny`	3.3.6	794	5	66	232	1
`typesystem`	0.2.4	157	3	23	19	13
Total		12912	163	1586	1545	210

on Debian 10 together with Python 3.10.5 (the `python:3.10.5-slim-buster` image from Docker Hub is the basis of this container).

We ran Pynguin in ten different configurations: the five generation approaches *DynaMOSA*, *MIO*, *MOSA*, *Random*, and *WS* (see Section 3.1 for a description of each approach), each with and without incorporated type information. We will refer to a configuration with incorporated type information by adding the suffix *TypeHints* to its name, for example, *DynaMOSA-TypeHints* for DynaMOSA with incorporated type information; the suffix *NoTypes* indicates a configuration that omits type information, for example, *Random-NoTypes* for feedback-directed random test generation without type information.

We adopt several default parameter values from EvoSuite (Fraser and Arcuri 2013). It has been empirically shown (Arcuri and Fraser 2013) that these default values give reasonably acceptable results. We leave a detailed investigation of the influence of the parameter values as future work. The following values are only set if applicable for a certain configuration. For our experiments, we set the population size to 50. For the MIO algorithm, we cut the search budget to half for its exploration phase and the other half for its exploitation phase. In the exploration phase, we set the number of tests per target to ten, the probability to pick either a random test or from the archive to 50 %, and the number of mutations to one. For the exploitation phase, we set the number of tests per target to one, the probability to pick either a random test or from the archive to zero, and the number of mutations to ten. We use a single-point crossover with the crossover probability set to 0.750. Test cases

and test suite are mutated using uniform mutation with a mutation probability of $\frac{1}{l}$, where l denotes the number of statements contained in the test case. PYNGUIN uses tournament selection with a default tournament size of 5. The search budget is set to 600 s. This time does not include preprocessing and pre-analysis of the subject. Neither does it include any post-processing, such as assertion generation or writing test cases to a file. The search can stop early if 100 % coverage is achieved before consuming the full search budget. Please note that the stopping condition for the search budget is only checked between algorithm iterations. Thus, it may be possible that some executions of PYNGUIN slightly exceed the 600 s search budget before they stop the search.

To minimise the influence of randomness we ran PYNGUIN 30 times for each configuration and module. All experiments were conducted on dedicated compute servers each equipped with an AMD EPYC 7443P CPU and 256 GB RAM, running Debian 10. We assigned each run one CPU core and four gigabytes of RAM.

4.1.3 Evaluation Metrics

We use code coverage to evaluate the performance of the test generation in RQ1 and RQ2. In particular, we measure branch coverage at the level of Python bytecode. Branch coverage is defined as the number of covered, that is, executed branches in the subject under test divided by the total number of branches in the subject under test. Similar to Java bytecode, complex conditions are compiled to nested branches with atomic conditions in Python code. We also keep track of coverage over time to shed light on the speed of convergence of the test-generation process; besides, we note the final overall coverage.

To evaluate the quality of the generated assertions in RQ3 we compute the mutation score. We use a customised version of the MUTPY tool (Derezinska and Hałas 2014) to generate the mutants from the original subject under test. MUTPY brings a large selection of standard mutation operators but also operators that are specific to Python. A mutant is a variant of the original subject under test obtained by injecting artificial modifications into them. It is referred to as *killed* if there exists a test that passes on the original subject under test but fails on the mutated subject under test. The mutation score is defined as the number of killed mutants divided by the total number of generated mutants (Jia and Harman 2011).

We statistically analyse the results to see whether the differences between two different algorithms or configurations are statistically significant or not. We exclude modules from the further analysis for which PYNGUIN failed to generate tests 30 times in all respective configurations to make the configurations comparable. We use the non-parametric Mann-Whitney U-test (Mann and Whitney 1947) with a $p-$value threshold 0.050 0 for this. A $p-$value below this threshold indicates that the null hypothesis can be rejected in favour of the alternative hypothesis. In terms of coverage, the null hypothesis states that none of the compared algorithms reaches significantly higher coverage; the alternative hypothesis states that one of the algorithms reaches significantly

higher coverage values. We use the Vargha and Delaney effect size $\hat{A}_{12}$ (Vargha and Delaney 2000) in addition to testing for the null hypothesis. The effect size states the magnitude of the difference between the coverage values achieved by two different configurations. For equivalent coverage the Vargha-Delaney statistics is $\hat{A}_{12} = 0.500$. When comparing two configurations C_1 and C_2 in terms of coverage, an effect size of $\hat{A}_{12} > 0.500$ indicates that configuration C_1 yields higher coverage than C_2; vice versa for $\hat{A}_{12} < 0.500$. Furthermore, we use Pearson's correlation coefficient r (Pearson 1895) to measure the linear correlation between two sets of data. We call a value of $r = \pm 1$ a *perfect correlation*, a value of r between ± 0.500 and ± 1 a *strong correlation*, a value of r between ± 0.300 and ± 0.499 a *medium correlation*, and a value of r between ± 0.100 and ± 0.299 a *small* or *weak correlation*.

Finally, please note that we report all numbers during our experiments rounded to three significant digits, except if they are countable, such as, for example, lines of code in a module.

4.1.4 Data Availability

We make all used projects and tools as well as the experiment and data-analysis infrastructure together with the raw data available on Zenodo (Lukasczyk 2022). This shall allow further analysis and replication of our results.

4.2 Threats to Validity

As usual, our experiments are subject to a number of threats to validity.

4.2.1 Internal Validity

The standard coverage tool for Python is Coverage.py, which offers the capability to measure both line and branch coverage. It, however, measures branch coverage by comparing the transitions between sources lines that have occurred and that are possible. Measuring branch coverage using this technique is possibly imprecise. Not every branching statement necessarily leads to a source line transition, for example, `x = 0 if y > 42 else 1337` fits on one line but contains two branches, which are not considered by Coverage.py. We thus implemented our own coverage measurement based on bytecode instrumentation. By providing sufficient unit tests for it we try to mitigate possible errors in our implementation.

Similar threats are introduced by the mutation-score computation. A selection of mutation-testing tools for Python exist, however, each has some individual drawbacks, which make them unsuitable for our choice. Therefore, we implemented the computation of mutation scores ourselves. However, the mutation of the subject under test itself is done using a customised version of the MutPy mutation testing tool (Derezinska and Hałas 2014) to better control this threat.

A further threat to the internal validity comes from probably flaky tests; a test is flaky when its verdict changes non-deterministically. Flakiness is reported to be a problem, not only for Python test suites (Gruber et al. 2021) but also for automatically generated tests in general (Fan 2019; Parry et al. 2022).

The used Python inspection to generate the test cluster (see Section 3.3) cannot handle types provided by native dependencies. We mitigate this threat by excluding projects that have dependencies with native code. This, however, does not exclude any functions from the Python standard library, which is partially also implemented in C, and which could influence our results.

4.2.2 External Validity

We used 163 modules from different Python projects for our experiments. It is conceivable that the exclusion of projects without type annotations or native-code libraries leads to a selection of smaller projects, and the results may thus not generalise to other Python projects. Furthermore, to make the different configurations comparable, we omitted all modules from the final evaluation for that PYNGUIN was not able to generate test cases for each configuration and each of the 30 iterations. This leads to 134 modules for RQ1 and RQ2 and 105 modules for RQ3. The number of used modules for RQ3 is lower because we exclude modules from the analysis that did not yield 30 results. Reasons for such failures are, for example, flaky tests. However, besides the listed constraints, no others were applied during this selection.

4.2.3 Construct Validity

Methods called with wrong input types still may cover parts of the code before possibly raising exceptions due to the invalid inputs. We conservatively included all coverage in our analysis, which may improve coverage for configurations that ignore type information. A configuration that does not use type information will randomly pick types to generate argument values, although these types might be wrong. In contrast, configurations including type information will attempt to generate the correct type; they will only use a random type with small probability. Thus, this conservative inclusion might reduce the effect we observed. It does, however, not affect our general conclusions.

Additionally, we have not applied any parameter tuning to the search parameters but use default values, which have been shown to be reasonable choices in practice (Arcuri and Fraser 2013).

4.3 RQ1: Comparison of the Test-Generation Approaches

The violin plots in Fig. 3 show the coverage distributions for each algorithm. We use all algorithms here in a configuration that incorporates type information. We note coverage values over the full range of 0 % to 100 %. Notably, all violins show a higher coverage density above 20 %, and very few modules result in lower

coverage; this is caused by what we call import coverage. Import coverage is achieved by importing the module; when Python imports a module it executes all statements at module level, such as imports, or module-level variables. It also executes all function definitions (the `def` statement but not the function's body or any closures) as well as class definitions and their respective method definitions. Due to the structure of the Python bytecode these definitions are also (branchless) coverage targets that get executed anyway. Thus, they count towards coverage of a module. As a consequence coverage cannot drop below import coverage.

The distributions for the different configurations look very similar, indicating a very similar performance characteristics of the algorithms; the notable exception is the Random algorithm with a lower performance compared to the evolutionary algorithms.

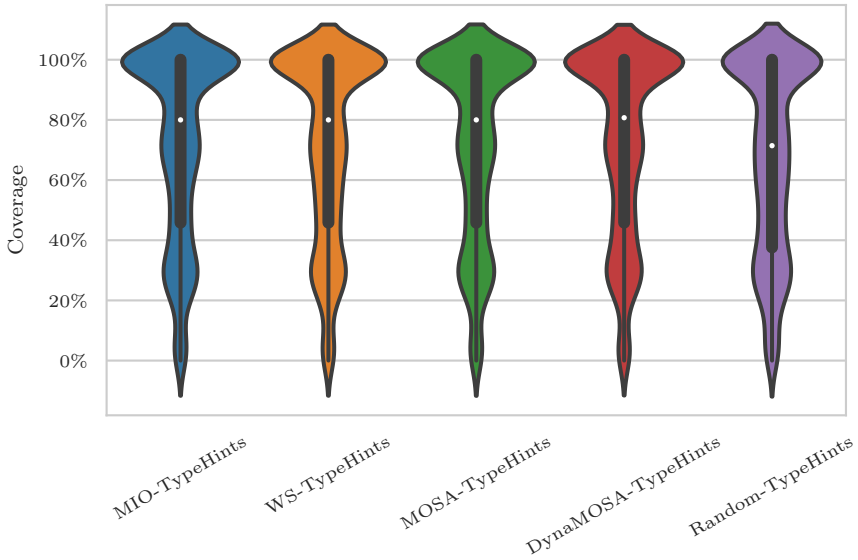

Fig. 3: Coverage distribution per algorithm with type information. The median value is indicated by a white dot within the inner quartile markers.

Although the violin plot reports the median values (indicated by a white dot), we additionally report the median and mean coverage values for each configuration in Table 2. The table shows the almost equal performance of the evolutionary algorithms DynaMOSA, MIO, and MOSA. Our random algorithm achieves the lowest coverage values in this experiment.

Since those coverage values are so close together, we computed $\hat{A}_{12}$ statistics for each pair of DynaMOSA and one of the other algorithms on the coverage of

Table 2: Comparison of the different algorithms, with and without type information. The table shows the median coverage value, as well as the mean coverage value with standard deviation.

Algorithm	With Type Hints		Without Type Hints	
	median (%)	mean (%)	median (%)	mean (%)
DynaMOSA	80.7	71.6 ± 30.5	76.5	69.4 ± 31.0
MIO	80.0	71.3 ± 30.7	75.0	68.4 ± 31.3
MOSA	80.0	71.3 ± 30.8	76.7	68.7 ± 31.7
Random	71.4	66.9 ± 31.5	66.7	62.6 ± 32.9
WS	80.0	70.6 ± 30.7	71.4	67.5 ± 31.5

all modules. All effects are negligible but in favour of DynaMOSA (DynaMOSA and MIO: $\hat{A}_{12} = 0.502$; DynaMOSA and MOSA: $\hat{A}_{12} = 0.502$; DynaMOSA and Random: $\hat{A}_{12} = 0.541$; DynaMOSA and WS: $\hat{A}_{12} = 0.508$). The effects are not significant except for DynaMOSA and Random with $p = 1.08 \times 10^{-10}$. We also compared the effects on coverage on a module level. The following numbers report the count of modules where an algorithm performed significantly better or worse than DynaMOSA. MIO performed better than DynaMOSA on 3 modules but worse on 13. MOSA performed better than DynaMOSA on 1 module but worse on 5. Random performed better than DynaMOSA on 0 modules but worse on 52. Whole Suite performed better than DynaMOSA on 1 module but worse on 25. We see that although modules exist where other algorithms outperform DynaMOSA significantly, overall DynaMOSA performs better than the other algorithms.

We now show the development of the coverage over the full generation time of 600 s. The line plot in Fig. 4 reports the mean coverage values per configuration measured in one-second intervals. We see that during the first minute, MIO yields the highest coverage values before DynaMOSA is able to overtake MIO, while MOSA can come close. However, the performance of MIO decreases over the rest of the exploration phase. From the plot we can see that MOSA comes close to MIO at around 300 s. At this point, MIO switches over to its exploitation phase, which again seems to be beneficial compared to MOSA. Over the full generation time, Whole Suite yields smaller coverage values than the previous three, as does Random.

We hypothesize that the achieved coverage is influenced by properties of the module under test. Our first hypothesis is that there exists some correlation between the achieved coverage on a module and the number of lines of code in that module. In order to study this hypothesis, we use our best-performing algorithm, DynaMOSA, and compare the mean coverage values per module with the lines of code in the module. The scatter plot in Fig. 5 shows the result; we fitted a linear regression line in red colour into this plot. The data shows a weak negative correlation (Pearson $r = -0.211$ with a $p-$value of 0.014 3, which indicates that there is at least some support for this hypothesis: it is slightly easier to achieve higher coverage values on modules with fewer lines of

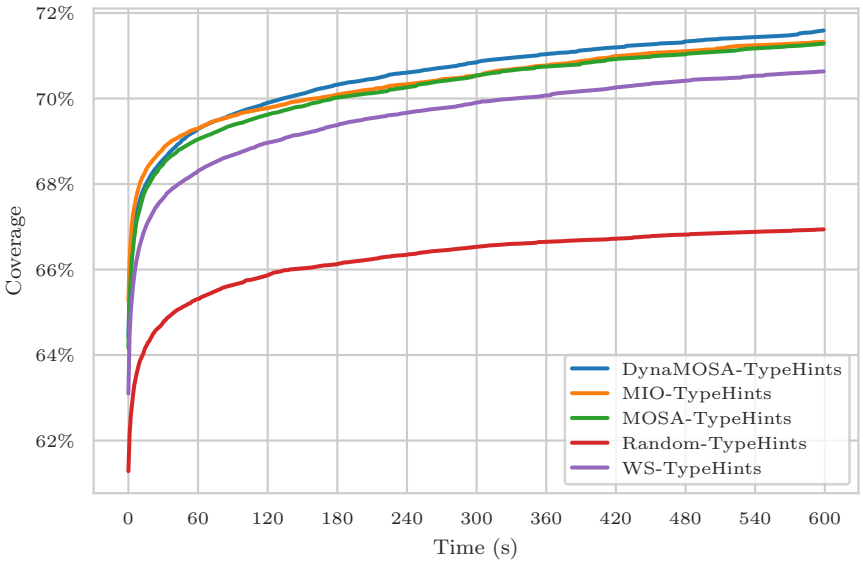

Fig. 4: Development of the coverage over time with available type information.

code. However, lines of code is often not considered a good metric to estimate code complexity. Therefore, we similarly study the correlation of mean coverage values per module with the mean McCabe cyclomatic complexity (McCabe 1976) of that module. The scatter plot in Fig. 6 shows the results; again, we fitted a linear regression line in red colour into the plot. The data shows a medium negative correlation (Pearson $r = -0.365$ with a $p-$value of 3.00×10^{-5}), supporting this hypothesis: modules with higher mean McCabe cyclomatic complexity tend to be more complicated to cover. However, since this correlation still is not strong, other properties of a module appear to influence the achieved coverage. Possible properties might be the quality of available type information or the ability of the test generator to instantiate objects of requested types properly. Also finding appropriate input values for function parameters might influence the achievable coverage. We study the influence of type information in RQ2, and leave exploring further factors as future work.

Summary (RQ1)

Our experiments show that test-generation algorithms in Python yield reasonable coverage values between 66.9 % to 71.6 % in the mean. Furthermore, we show that DynaMOSA performed best in this experiment, followed by MIO, MOSA, and Whole Suite; Random achieves the least coverage values.

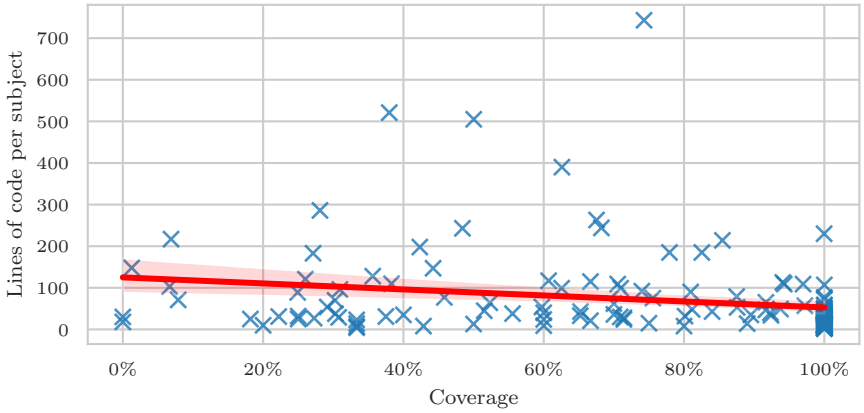

Fig. 5: Mean coverage for DynaMOSA with type information correlated to the lines of code per module; the red line shows the linear regression fitted to the data (Pearson $r = -0.211$, $p = 0.014\,3$).

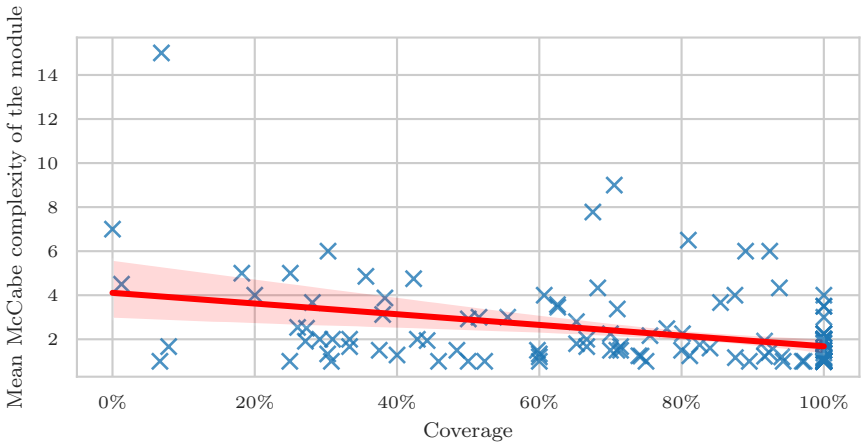

Fig. 6: Mean coverage for DynaMOSA with type information correlated to the mean McCabe cyclomatic complexity of a module's functions; the red line shows the linear regression fitted to the data (Pearson $r = -0.365$, $p = 3.00 \times 10^{-5}$).

Discussion: Overall, the results we achieved from our experiments indicate that automated test generation for Python is feasible. They also show that there exists a ranking between the performance of the studied algorithms, which is in line with previous research on test generation in Java (Campos et al. 2018).

The results, however, do not show large differences between the algorithms; only DynaMOSA compared to Random yielded a significant, although negligible, effect. A reason, why the algorithms have very similar performance is indicated by the subjects that we use for our study. We noticed subjects that are trivial to cover for all algorithms. Listing 9 shows such an example taken from the module `flutes.math` from the `flutes` project[12]. Almost every pair of two integers is

Listing 9: A function that can be trivially covered, taken from `flutes.math`.

```
def ceil_div(a: int, b: int) -> int:
    r"""Integer division that rounds up."""
    return (a - 1) // b + 1
```

a valid input for this function (only `b = 0` will cause a `ZeroDivisionError`). Since this function is the only function in that particular module, achieving full coverage on this module is also trivially possible for all test-generation algorithms, especially since they know from the type hints that they shall provide integer values as parameters.

Another category of modules that is hard to cover, independent of the used algorithm, is due to technical limitations of Pynguin (see Section 3.8). Consider the minimised code from the `flutes`[13] project in Listing 10. This

Listing 10: A function that is actually a context manager and a generator and thus cannot be covered by Pynguin due to technical limitations, taken from `flutes.timing`.

```
import contextlib
import time

@contextlib.contextmanager
def work_in_progress(desc: str = "Work in progress"):
    print(desc + "... ", end='', flush=True)
    begin_time = time.time()
    yield
    time_consumed = time.time() - begin_time
    print(f"done. ({time_consumed:.2f}s)")
```

function is actually both a context manager (due to its decorator) and a generator (indicated by the `yield` statement). Pynguin currently supports neither; the context manager would require a special syntax construct to be called, such as a `with` block. Calling a function with a `yield` statement in it does not actually execute its code. It generates an iterator object pointing to

[12] `https://www.pypi.org/project/flutes`, accessed 2022–07–14.
[13] `https://www.pypi.org/project/flutes`, accessed 2022–07–14.

that function. The code will only be executed when iterating over the generator in a loop or by explicitly calling `next` on the object. A test case PYNGUIN can come up with is similar to the one shown in Listing 11. This test case would

Listing 11: A test case from PYNGUIN for the function in Listing 10.

```
import flutes.timing as module_0

def test_case_0:
    generator_context_manager_0 = module_0.work_in_progress()
    assert generator_context_manager_0.args == ()
    assert generator_context_manager_0.kwds == {}
```

only result in a coverage of 50 %, which is only import coverage resulting from executing the `import` and `def` statements during module loading. As stated above, the body of the function will not even be executed.

Figure 5 and Fig. 6 indicate that our subject modules are not very complex. Previous research has shown that algorithms like DynaMOSA or MIO are more beneficial for a large number of goals, that is, a large number of branches (Panichella et al. 2015; Arcuri 2018). For modules with only few branches they cannot show their full potential which definitely influences our results. Having smaller modules is a property of our evaluation set. On average, our modules consist of 79.2 lines of code; Mir et al. (2021) report an average module size of 120 lines of code. Future work shall repeat our evaluation using more complex subject systems in order to evaluate whether the assumed improvements can be achieved there.

4.4 RQ2: Influence of Type Information

We hypothesized in the previous section that type information might have an impact on the achieved coverage. We compare the configuration with type information and the configuration without type information for each algorithm. Our comparison is done on the module level.

We plot the effect-size distributions per project for DynaMOSA, our best-performing algorithm from RQ1, in Fig. 7. We aggregate the data for a better overview here; we will also show the data on a module level afterwards. Each data point that is used for the plot is the effect size on one module of that project. The results show that the median effect is always greater or equal than 0.500. This entails that available type information is beneficial in the mean for four out of our twenty projects.

We do not only report the data on a project level as we did using Fig. 7 but also per module. We provide a table that reports the effect size per module in Table 3 in the appendix. A value larger than 0.500 for a module indicates that the configuration with type information yielded higher coverage results

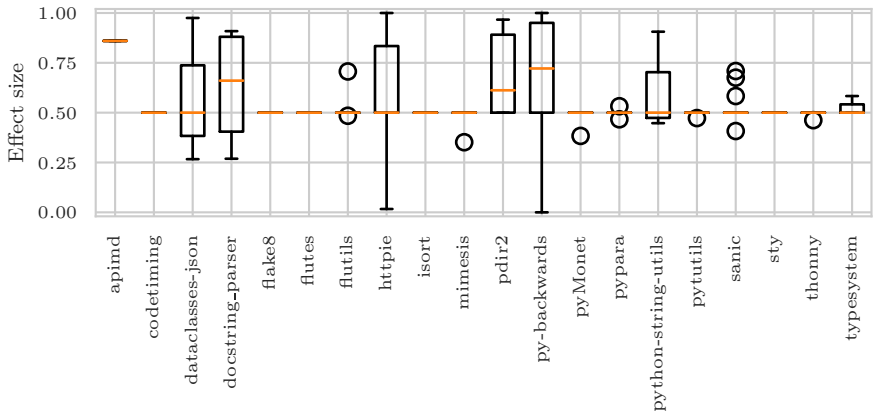

Fig. 7: Effect-size distributions per project for DynaMOSA. $\hat{A}_{12} > 0.500$ indicates that DynaMOSA-TypeHints performed better than DynaMOSA-NoTypeHints; vice versa for $\hat{A}_{12} < 0.500$.

than the configuration without type information. A value smaller than 0.500 indicates the opposite. We use a bold font to mark a table entry where the effect size is significant with respect to a $p-$value < 0.05.

Summary (RQ2)

Our experiment shows that type information can be beneficial for the effectiveness of the test-generation algorithms. The effect, however, largely depends on the module under test.

Discussion: Our results show that type information can be beneficial to achieve higher coverage, although this largely depends on the module under test. From Table 3 we also note that there are cases where the presence of type information has a negative impact compared to its absence.

A reason for this is that the type information is annotated manually by the developers of the projects. Although powerful static type checking tools, such as MyPy are available, they are often not used. Rak-amnouykit et al. (2020) conducted a study using more than 2 600 repositories consisting of more than 173 000 files that have partial type annotations available. They report that often type information is only partially available. As a consequence, large parts of the code still do not have any type information at all. They furthermore report that the type annotations are often wrong, too: they were able to correctly type check only about 11.9 % of the repositories using the MyPy tool. Such wrong or incomplete type information can mislead Pynguin as to which parameter types to use. For wrong type information there is still a small probability for

Pynguin to disregard the type hint anyway and use a random type. The consequence for missing type information is that Pynguin needs to use a random type, anyway.

We also noticed cases where parts of the code can only be executed if one uses a type for an argument that is different from the annotated type. An example can be error-handling code that explicitly raises an error in case the argument type is wrong. We depict a simple example of such a case in Listing 12. In this example, we see that the then-branch of the `isinstance` check can only

Listing 12: A function having a precondition. The then-branch can only be executed if one uses a type for the argument that is different from the annotated type.

```
def example(a: int) -> int:
    if isinstance(a, str):
        raise TypeError("Expected an int but not a string")
    return 2 * a
```

be executed if the argument to the function is of type `str`—in contradiction to the annotated `int`. Although it is possible that Pynguin disregards the type hint there is only a small probability for this. Besides, Pynguin would then pick a random type from all available types in the test cluster, where again the probability that it picks a `str` is small, depending on the size of the test cluster.

Yet another reason for the results lies in the limitations Pynguin currently has with respect to generating objects of specific characteristics. Pynguin is, for example, not able to provide generators or iterators as arguments; neither does is provide higher-order functions. Adding these features is open as future work. Higher-order functions are required as an argument to 49 functions throughout the 134 modules. Being able to generate higher-order functions in the context of a dynamically-typed programming language has also been shown to be beneficial (Selakovic et al. 2018); we leave this for future work.

4.5 RQ3: Assertion Effectiveness to Reveal Faults

For our last research question we study the assertion effectiveness. To measure the effectiveness of the generated assertions we use the mutation-score metric. We limit our presentation of results for this questions to a smaller subset of our modules: 105 modules that we ran with DynaMOSA-TypeHints, our best-performing configuration with respect to achieved coverage. The remaining modules from the original 134 caused various issues and failed to be yield results for all 30 reruns. Please note that for answering RQ1 and RQ2, we configured Pynguin in a way that it does not create assertions but only reaches for coverage. We did this both to save computation time as well as to

prevent Pynguin from failing on modules due to issues related to the assertion generation, such as flaky tests. The issues that cause a drop in the number of modules for answering this research question were caused by our used mutation approach, flakiness, or incorrect programs and non-terminating loops.

Figure 8 reports the distribution of the mutation scores for this configuration in a violinplot.

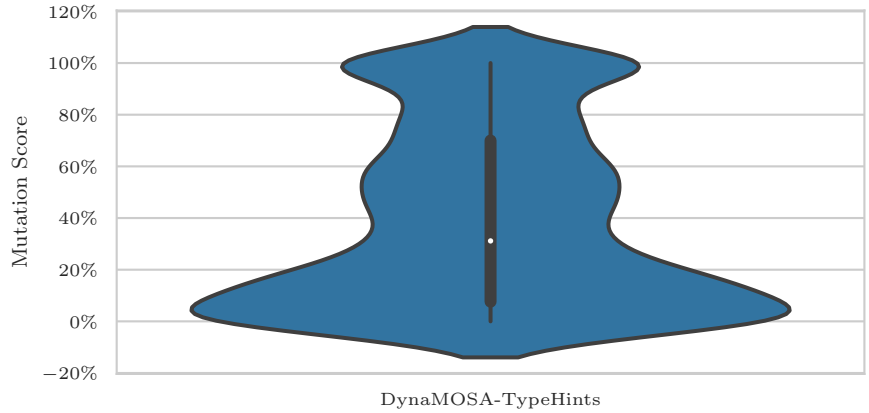

Fig. 8: A violinplot showing the distribution of the mutation scores for the used DynaMOSA configuration in our experiment for RQ3.

Additionally, we hypothesize that higher mutation scores are correlated with higher coverage, as it seems unlikely that for modules with small coverage values we can achieve high mutation scores. The rationale behind this hypothesis is that only mutated statements of a program that are covered by the execution can be detected. For this, we plot the achieved coverage values and mutation scores into a scatter plots; Fig. 9 shows these results. Besides, we add the number of lines of code per module as an additional dimension to the plots, visualised by the hue; a lighter colour stands for few lines of code, while a dark colour symbolises a module with a large number of code lines.

The plot shows a strong correlation between coverage and mutation score, with Pearson's correlation coefficient being computed to be $r = 0.450$ with a $p-$value of 5.53×10^{-157} Interestingly, there exist a few outliers where mutation score is 100 % although the coverage is only at a mediocre level of around 30 %. We investigated these cases and found out that mutants were only introduced in those parts of the module that were also covered by the test case. We conjecture this is caused by the mutation code which our implementation uses from the MutPy library, in combination with limitations of Pynguin. An example is the simplified snippet from `pyMonet`'s[14] module `pymonet.task`, which we show

[14] `https://pypi.org/project/pyMonet`, accessed 2022–07–15.

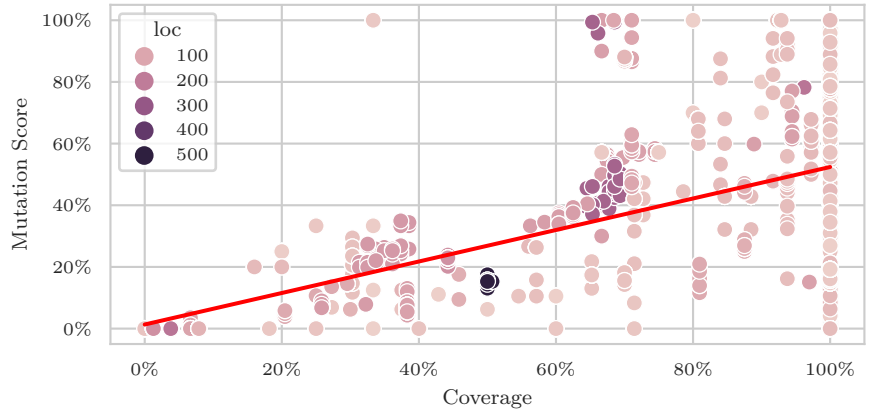

Fig. 9: Correlation between coverage and mutation score for DynaMOSA-TypeHints (Pearson $r = 0.450$, $p = 5.53 \times 10^{-157}$).

in Listing 13. The module consists of 15 branchless code objects, each of which is a coverage goal[15]. Of these, Pynguin is able to cover only five (the module, the class declaration, the constructor declaration, and the declarations of the `map` and `bind` methods). The methods `of` and `reject` are not even called as they are annotated as class methods, which is currently not supported by Pynguin. Pynguin, however, is generating a test cases like the one shown in Listing 14. When mutating the module from Listing 13, the only possible change is to remove a `@classmethod` decorator. The consequence is that `of` or `reject` become normal methods by this operation. Executing the test case from Listing 14 against such a mutant causes a difference in the result and thus Pynguin counts the mutant as killed. As a consequence, both possible mutants get killed, although only one third of the modules coverage goals are actually covered. Furthermore, we also note many modules for those the tests achieve 100 % coverage but the achieved mutation scores range from 0 % to 100 %; this shows that the effectiveness of our assertions can still be improved in the future.

Summary (RQ3)

Our experiment shows that Pynguin is able to generate assertions of good effectiveness—they can achieve mutation scores of up to 100 %.

Discussion: Our experiment indicates that Pynguin is able to provide high-quality assertions. As we expected, we note a strong correlation between

[15] The module itself, the class, each method, each closure, and each lambda form such code objects.

Listing 13: Simplified snipped from `pymonet.task` showing different results in the original version than in a mutant

```
class Task:
  def __init__(self, fork):
    self.fork = fork

  @classmethod
  def of(cls, value):
    return Task(lambda _, resolve: resolve(value))

  @classmethod
  def reject(cls, value):
    return Task(lambda reject, _: reject(value))

  def map(self, fn):
    def result(reject, resolve):
      return self.fork(lambda arg: reject(arg), lambda arg:
          resolve(fn(arg)))
    return Task(result)

  def bind(self, fn):
    def result(reject, resolve):
      return self.fork(lambda arg: reject(arg), lambda arg: fn(arg
          ).fork(reject, resolve))
    return Task(result)
```

Listing 14: A test cases for the snippet in Listing 13.

```
import pymonet.task as module_0

def test_case_0:
  int_0 = 317
  task_0 = module_0.Task(int_0)
  assert module_0.Task.of is not None
  assert module_0.Task.reject is not None
```

coverage and mutation score (see Fig. 9). Using mutation score as an indicator for fault-finding capabilities is backed by the literature (see, for example, the work of Just et al. (2014) or Papadakis et al. (2018)). However, it is still open in the literature whether there exists a strong correlation between mutation scores and fault detection.

To answer this research question we had to shrink the set of subject modules from 134 down to 105. This is caused by various influences: we use a customised version of MutPy (Derezinska and Hałas 2014) to generate the mutants (see Section 3.7). The original MutPy has not received an update since 2019 and is designed to work with Python versions 3.4 to 3.7[16]; Pynguin, however, requires Python version 3.10 for our experiments. Several Python

[16] `https://github.com/mutpy/mutpy`, accessed 2022–07–12.

internals, such as the abstract syntax tree (AST) that is used by MutPy to mutate the original module, received changes between Python 3.7 and 3.10. Although we fixed the obvious issues, our customised version of MutPy is an issue for crashes: future work shall replace this component by our own implementation of the various mutation operators for a more fine-grained control.

Furthermore, mutation at the AST level can cause various problems, for example, incorrect programs or non-terminating loops. We tried to make our test execution as robust against non-termination as we could, still we noted that there are open issues the experiment infrastructure to shutdown Pynguin hardly after a walltime of 2.50 h—without reporting any results.

In some cases already the original subject module shows non-deterministic behaviour for example, due to the use of random numbers. Our assertion generation executes each generated test case twice in random order to remove trivially flaky assertions. Previous research (Gruber et al. 2021), however, has shown that in order to achieve a 95 % confidence that a passing test case is not flaky will require 170 reruns on average. Obviously, those reruns would consume a huge amount of time, which Pynguin should not invest into rerunning tests to potentially find a flaky one. Therefore, we accept that Pynguin generates flaky tests, which can influence the assertion generation and cause failures.

The results we report here, however, are promising. They show that also mutation-based assertion generation is feasible for dynamically typed programming languages; this again opens up new research perspectives targeting the underlying oracle problem.

5 Related Work

Closest to our work is EvoSuite (Fraser and Arcuri 2013), which is a unit test generation framework for Java. Similar to our approach, EvoSuite incorporates many different test-generation strategies to allow studies on their different behaviour and characteristics.

To the best of our knowledge, little has been done in the area of automated test generation for dynamically typed languages. Search-based test generation tools have been implemented before, for example, for the Lua programming language (Wibowo et al. 2015) or Ruby (Mairhofer et al. 2011). While these approaches utilise a genetic algorithm, they are only evaluated on small data sets and do not provide comparisons between different test-generation techniques.

Approaches such as SymJS (Li et al. 2014) or JSEFT (Mirshokraie et al. 2015) target specific properties of JavaScript web applications, such as the browser's DOM or the event system. Feedback-directed random testing has also been adapted to web applications with Artemis (Artzi et al. 2011). Recent work proposes LambdaTester (Selakovic et al. 2018), a test generator that specifically addresses the generation of higher-order functions in dynamic languages. Our approach, in contrast, is not limited to specific application domains.

A related approach to our random test generation can be found in the Hypothesis library, a library for property-based testing in Python (MacIver and Hatfield-Dodds 2019; MacIver and Donaldson 2020). Hypothesis uses a random generator to generate the inputs for checking whether a property holds. It is also able to generate and export test cases based on this mechanism. It does, however, only implement a random generator and does not focus on unit-test generation primarily. TSTL (Groce and Pinto 2015; Holmes et al. 2018) is another tool for property-based testing in Python although it can also be used to generate unit tests. It utilises a random test generator for this but its main focus is not to generate unit tests automatically, which we aim to achieve with our Pynguin approach.

Further tools are, for example, Auger[17], CrossHair[18], or Klara[19]; they all require manual effort by the developer to create test cases in contrast to our automated generation approach.

Additionally, dynamically typed languages have also gained attention from research on test amplification. Schoofs et al. (2021) introduce PyAmplifier a proof-of-concept tool for test amplification in Python. Recently, SmallAmp (Abdi et al. 2022) demonstrated the use of dynamic type profiling as a substitute for type declarations in the Pharo Smalltalk language.

In recent work, Pynguin has also been evaluated on a scientific software stack from material sciences (Trübenbach et al. 2022). The authors also compared the performance of different test-generation algorithms and the assertion generation on their subject system; their results regarding the algorithm performance are in line with our results, whereas their results on assertion quality differ significantly due to their use of an older version of Pynguin that only provided limited assertion-generation capabilities.

6 Conclusions

In this work, we presented Pynguin, an automated unit test generation framework for Python. We extended our previous work (Lukasczyk et al. 2020) by incorporating the DynaMOSA, MIO, and MOSA algorithms, and by evaluating the achieved coverage on a larger corpus of Python modules. Our experiments demonstrate that Pynguin is able to emit unit tests for Python that cover large parts of existing code bases. In line with prior research in the domain of Java unit test generation, our evaluation results show that DynaMOSA performs best in terms of branch coverage, followed by MIO, MOSA, and the Whole Suite approach.

While our experiments provide evidence that the availability of type information influences the performance of the test-generation algorithms, they also show that there are several open issues that provide opportunities for further research. A primary challenge is that adequate type information may

[17] `https://github.com/laffra/auger`, accessed 2022–07–12.
[18] `https://crosshair.readthedocs.io`, accessed 2022–02–10,
[19] `https://klara-py.readthedocs.io`, accessed 2022–02–10,

not be available in the first place, suggesting synergies with research on type inference. However, even if type information is available, generating instances of the relevant types remains challenging. There are also technical challenges that PYNGUIN needs to address in order to become practically usable; overcoming technical limitations is necessary to allow the usage of PYNGUIN for a wider field of projects and scenarios.

While we were investigating in the technical limitations of PYNGUIN, we also found a real bug in one of our subjects: the module `mimesis.providers.date` from the `mimesis`[20] contains the following function presented in Listing 15 (we removed various lines that are not relevant to the bug for presentation reasons).

Listing 15: Simplified code snippet taken from `mimesis.providers.date`

```
from datetime import datetime, timedelta

def bulk_create_datetimes(
    date_start: datetime, date_end: datetime, **kwargs: Any
) -> list[datetime]:
    dt_objects = []

    if date_end < date_start:
        raise ValueError()

    while date_start <= date_end:
        date_start += timedelta(**kwargs)
        dt_objects.append(date_start)

    return dt_objects
```

The `mimesis` project provides utility functions that generate fake data, which aims to look as realistic as possible. The function presented in Listing 15 aims to generate a list of `datetime` objects (essentially, a combination of date and time stamp) between a given start and end time using a specific interval. It requires a start time that is smaller than the end time, that is, lies before the end time. Besides, it takes a dictionary in its `**kwargs` parameter, which it hands over to the `timedelta` function of Python's `datetime` API.

The `timedelta` function is used to compute a delta that can be added to the current time in order to retrieve the next date. Its API is very flexible: it allows to specify an arbitrary delta, especially also of negative and zero values. While we were investigating in PYNGUIN's timeout handling, we noticed that PYNGUIN generated a test case for this function similar to the one presented in Listing 16. Executing this test case leads to an infinite loop because the `timedelta` function will yield a delta of zero if no parameters given.

[20] `https://pypi.org/project/mimesis`, accessed 2022–07–14.

Listing 16: A bug-exposing test case

```
import datetime
import mimesis.providers.date as module_0

def test_case():
    int_0 = 2022
    int_1 = 1
    int_2 = 2
    date_time_0 = datetime.datetime(int_0, int_1, int_1)
    date_time_1 = datetime.datetime(int_0, int_1, int_2)
    module_0.bulk_create_datetimes(date_time_0, date_time_1)
```

We reported this issue to the developers of `mimesis`, who stated that this 'is a major bug actually'[21]. They furthermore accepted the fix proposed by the second author of this paper and released a new version of their library.

More information on PYNGUIN is available from its web page:

`https://www.pynguin.eu`

Conflict of interest

The authors declare that they have no conflict of interest.

References

Abdi M, Rocha H, Demeyer S, Bergel A (2022) Small-amp: Test amplification in a dynamically typed language. Empirical Software Engineering 27(128), DOI 10.1007/s10664-022-10169-8

Acree AT, Budd TA, DeMillo RA, Lipton RJ, Sayward FG (1978) Mutation analysis. Tech. Rep. GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia, USA

Andrews JH, Menzies T, Li FCH (2011) Genetic algorithms for randomized unit testing. IEEE Transactions on Software Engineering 37(1):80–94, DOI 10.1109/TSE.2010.46

Arcuri A (2013) It really does matter how you normalize the branch distance in search-based software testing. Journal of Software Testing, Verification and Reliability 23(2):119–147, DOI 10.1002/stvr.457

Arcuri A (2017) Many independent objective (MIO) algorithm for test suite generation. In: International Symposium on Search Based Software Engineering (SSBSE), Springer, Lecture Notes in Computer Science, vol 10452, pp 3–17, DOI 10.1007/978-3-319-66299-2_1

Arcuri A (2018) Test suite generation with the many independent objective (MIO) algorithm. Information & Software Technology 104:195–206, DOI 10.1016/j.infsof.2018.05.003

Arcuri A, Fraser G (2013) Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering 18(3):594–623, DOI 10.1107/s10664-013-9249-9

Artzi S, Dolby J, Jensen SH, Møller A, Tip F (2011) A framework for automated testing of JavaScript web applications. In: International Conference on Software Engineering (ICSE), ACM, pp 571–580, DOI 10.1145/1985793.1985871

[21] `https://github.com/lk-geimfari/mimesis/pull/1229#issuecomment-1162974494`, accessed 2022–07–14.

Baresi L, Miraz M (2010) Testful: Automatic unit-test generation for java classes. In: International Conference on Software Engineering (ICSE), ACM, vol 2, pp 281–284, DOI 10.1145/1810295.1810353

Campos J, Ge Y, Albunian N, Fraser G, Eler M, Arcuri A (2018) An empirical evaluation of evolutionary algorithms for unit test suite generation. Information & Software Technology 104:207–235, DOI 10.1016/j.infsof.2018.08.010

Chen Z, Ma W, Lin W, Chen L, Li Y, Xu B (2018) A study on the changes of dynamic feature code when fixing bugs: Towards the benefits and costs of python dynamic features. SCIENCE CHINA Information Sciences 61(1):012107:1–012107:18, DOI 10.1007/s11432-017-9153-3

Csallner C, Smaragdakis Y (2004) Jcrasher: an automatic robustness tester for java. Software: Practice and Experience 34(11):1025–1050, DOI 10.1002/spe.602

DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: Help for the practicing programmer. Computer 11(4):34–41, DOI 10.1109/C-M.1978.218136

Derezinska A, Hałas K (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, pp 156–164, DOI 10.1109/ICSTW.2014.24

Fan Z (2019) A systematic evaluation of problematic tests generated by EvoSuite. In: International Conference on Software Engineering Companion (ICSE Companion), IEEE/ACM, pp 165–167, DOI 10.1109/ICSE-Companion.2019.00068

Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9(3):319–349, DOI 10.1145/24039.24041

Fraser G, Arcuri A (2012) The seed is strong: Seeding strategies in search-based software testing. In: International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, pp 121–130, DOI 10.1109/ICST.2012.92

Fraser G, Arcuri A (2013) Whole test suite generation. IEEE Transactions on Software Engineering 39(2):276–291, DOI 10.1109/TSE.2012.14

Fraser G, Zeller A (2012) Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering 38(2):278–292, DOI 10.1109/TSE.2011.93

Gao Z, Bird C, Barr ET (2017) To type or not to type: Quantifying detectable bugs in javascript. In: International Conference on Software Engineering (ICSE), IEEE/ACM, pp 758–769, DOI 10.1109/ICSE.2017.75

Gong L, Pradel M, Sridharan M, Sen K (2015) DLint: dynamically checking bad coding practices in JavaScript. In: International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 94–105, DOI 10.1145/2771783.2771809

Groce A, Pinto J (2015) A little language for testing. In: NASA International Symposium on Formal Methods (NFM), Springer, Lecture Notes in Computer Science, vol 9058, pp 204–218, DOI 10.1007/978-3-319-17524-9_15

Gruber M, Lukasczyk S, Kroiß F, Fraser G (2021) An empirical study of flaky tests in python. In: International Conference on Software Testing, Verification and Validation (ICST), IEEE, pp 148–158, DOI 10.1109/ICST49551.2021.00026

Holmes J, Groce A, Pinto J, Mittal P, Azimi P, Kellar K, O'Brien J (2018) TSTL: the template scripting testing language. International Journal on Software Tools for Technology Transfer 20(1):57–78, DOI 10.1007/s10009-016-0445-y

Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5):649–678, DOI 10.1109/TSE.2010.62

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) Are mutants a valid substitute for real faults in software testing? In: International Symposium on Foundations of Software Engineering (FSE), ACM, pp 654–665, DOI 10.1145/2635868.2535929

Kleinschmager S, Hanenberg S, Robbes R, Éric Tanter, Stefik A (2012) Do static type systems improve the maintainability of software systems? an empirical study. In: International Conference on Program Comprehension (ICPC), IEEE Computer Society, pp 153–162, DOI 10.1109/ICPC.2012.6240483

Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady, vol 10, pp 707–710

Li G, Andreasen E, Ghosh I (2014) SymJS: Automatic symbolic testing of JavaScript web applications. In: International Symposium on Foundations of Software Engineering (FSE), ACM, pp 449–459, DOI 10.1145/2635858.2635913

Lukasczyk S (2022) Artefact to our paper "an empirical study of automated unit test generation for python". DOI 10.5281/zenodo.6838658

Lukasczyk S, Fraser G (2022) Pynguin: Automated unit test generation for python. In: International Conference on Software Engineering Companion (ICSE Companion), ACM, pp 168–172, DOI 10.1145/3510454.3516829, to appear

Lukasczyk S, Kroiß F, Fraser G (2020) Automated unit test generation for python. In: International Symposium on Search Based Software Engineering (SSBSE), Springer, Lecture Notes in Computer Science, vol 12420, pp 9–24, DOI 10.1007/978-3-030-59762-7_2

Lukasczyk S, Kroiß F, Fraser G, Contributors P (2022) se2p/pynguin: Pynguin 0.25.2. DOI 10.5281/zenodo.6836225

von Lücken C, Barán B, Brizuela CA (2014) A survey on multi-objective evolutionary algorithms for many-objective problems. Computational Optimization and Applications 58(3):707–756, DOI 10.1007/s10589-014-9644-1

Ma L, Artho C, Zhang C, Sato H, Gmeiner J, Ramler R (2015) GRT: program-analysis-guided random testing (T). In: International Conference on Automated Software Engineering (ASE), IEEE Computer Society, pp 212–223, DOI 10.1109/ASE.2015.49

MacIver D, Donaldson AF (2020) Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In: European Conference on Object-Oriented Programming (ECOOP), Schloss Dagstuhl – Leibnitz-Zentrum für Informatik, Leibnitz International Proceedings in Informatics (LIPIcs), vol 166, pp 13:1–13:27, DOI 10.4230/LIPIcs.ECOOP.2020.13

MacIver D, Hatfield-Dodds Z (2019) Hypothesis: A new approach to property-based testing. Journal of Open Source Software 4(43):1891, DOI 10.21105/joss.01891

Mairhofer S, Feldt R, Torkar R (2011) Search-based software testing and test data generation for a dynamic programming language. In: Annual Conference on Genetic and Evolutionary Computation (GECCO), ACM, pp 1859–1866, DOI 10.1145/2001576.2001826

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The Annals of Mathematical Statistics 18(1):50–60, DOI 10.1214/aoms/1177730491

Mayer C, Hanenberg S, Robbes R, Éric Tanter, Stefik A (2012) An empirical study of the influence of static type systems on the usability of undocumented software. In: Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, pp 683–702, DOI 10.1145/2384616.2384666

McCabe TJ (1976) A complexity measure. IEEE Transactions on Software Engineering 2(4):308–320, DOI 10.1109/TSE.1976.233837

Meyerovich LA, Rabkin AS (2013) Empirical analysis of programming language adoption. In: Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, pp 1–18, DOI 10.1145/2509136.2509515

Mir AM, Latoskinas E, Gousios G (2021) Manytypes4py: A benchmark python dataset for machine learning-based type inference. In: IEEE Working Conference on Mining Software Repositories (MSR), IEEE, pp 585–589, DOI 10.1109/MSR52588.2021.00079

Mirshokraie S, Mesbah A, Pattabiraman K (2015) JSEFT: Automated javascript unit test generation. In: International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, pp 1–10, DOI 10.1109/ICST.2015.7102595

Pacheco C, Lahiri SK, Ernst MD, Ball T (2007) Feedback-directed random test generation. In: International Conference on Software Engineering (ICSE), IEEE Computer Society, pp 75–84, DOI 10.1109/ICSE.2007.37

Panichella A, Kifetew FM, Tonella P (2015) Reformulating branch coverage as a many-objective optimization problem. In: International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, pp 1–10, DOI 10.1109/ICST.2015.7102604

Panichella A, Kifetew FM, Tonella P (2018a) 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, DOI 10.1109/TSE.2017.2663435

Panichella A, Kifetew FM, Tonella P (2018b) A large scale empirical comparison of state-of-the-art search-based test case generators. Information & Software Technology 104:236–256, DOI 10.1016/j.infsof.2018.08.009

Papadakis M, Shin D, Yoo S, Bae D (2018) Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In: International Conference on Software Engineering (ICSE), ACM, pp 537–548, DOI 10.1145/3180155.3180183

Parry O, Kapfhammer GM, Hilton M, McMinn P (2022) A survey of flaky tests. ACM Transactions on Software Engineering and Methodology 31(1):17:1–17:74, DOI 10.1145/3476105

Pearson K (1895) Notes on regression and inheritance in the case of two parents. In: Proceedings of the Royal Society of London, vol 58, pp 240–242

Rak-amnouykit I, McCrevan D, Milanova AL, Hirzel M, Dolby J (2020) Python 3 types in the wild: A tale of two type systems. In: ACM SIGPLAN International Symposium on Dynamic Languages (DLS), ACM, pp 57–70, DOI 10.1145/3426422.3426981

Sakti A, Pesant G, Guéhéneuc Y (2015) Instance generator and problem representation to improve object oriented code coverage. IEEE Transactions on Software Engineering 41(3):294–313, DOI 10.1109/TSE.2014.2363479

Schoofs E, Abdi M, Demeyer S (2021) Ampyfier: Test amplification in python. CoRR abs/2112.11155, 2112.11155

Selakovic M, Pradel M, Karim R, Tip F (2018) Test generation for higher-order functions in dynamic languages. Proceedings of the ACM on Programming Languages 2(OOPSLA):161:1–161:27, DOI 10.1145/3276531

Tonella P (2004) Evolutionary testing of classes. In: International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 119–128, DOI 10.1145/1007512.1007528

Trübenbach D, Müller S, Grunske L (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: International Workshop on Search-Based Software Testing (SBST@ICSE)

Vargha A, Delaney HD (2000) A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25(2):101–132

Wappler S, Lammermann F (2005) Using evolutionary algorithms for the unit testing of object-oriented software. In: Annual Conference on Genetic and Evolutionary Computation (GECCO), pp 1053–1060, DOI 10.1145/1068009.1068187

Wibowo JTP, Hendradjaya B, Widyani Y (2015) Unit test code generator for lua programming language. In: International Conference on Data and Software Engineering (ICoDSE), IEEE, pp 241–245, DOI 10.1109/ICODSE.2015.7437005

Widyasari R, Sim SQ, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan JE, Yieh Y, Goh B, Thung F, Kang HJ, Hoang T, Lo D, Ouh EL (2020) BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In: Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 1556–1560, DOI 10.1145/3368089.3417943

Xie T (2006) Augmenting automatically generated unit-test suites with regression oracle checking. In: European Conference on Object-Oriented Programming (ECOOP), Springer, Lecture Notes in Computer Science, vol 4067, pp 380–403, DOI 10.1007/11785477_23

Xu Z, Liu P, Zhang X, Xu B (2016) Python predictive analysis for bug detection. In: International Symposium on Foundations of Software Engineering (FSE), ACM, pp 121–132, DOI 10.1145/2950290.2950357

A Comparison of Coverage Improvements

The following Table 3 presents the Vargha-Delaney $\hat{A}_{12}$ statistics for each module and algorithm on the resulting coverage values. A value larger than 0.500 indicates that the configuration with type information yielded higher coverage results than the configuration without type information; vice versa for values smaller 0.500. We use a **bold** font to indicate effect sizes that are significant.

Table 3: Improvement of types. The table shows the Vargha-Delaney $\hat{A}_{12}$ statistic for each module and DynaMOSA. A value larger than 0.500 indicates that the configuration with type information yielded higher coverage results than the configuration without type information; vice versa for values smaller 0.500. In **bold** font we denote those effects that are significant at a $p-$value $<$ 0.050 0.

Module Name	Effect Size
apimd.compiler	**0.860**
codetiming._timer	0.500
dataclasses_json.api	0.500
dataclasses_json.mm	**0.975**
dataclasses_json.undefined	**0.267**
docstring_parser.google	**0.909**
docstring_parser.numpydoc	**0.871**
docstring_parser.parser	0.450
docstring_parser.rest	**0.269**
flake8.exceptions	0.500
flake8.formatting.base	0.500
flake8.formatting.default	0.500
flake8.main.debug	0.500
flake8.main.git	0.500
flutes.math	0.500
flutes.timing	0.500
flutils.decorators	0.500
flutils.namedtupleutils	0.500
flutils.packages	0.484
flutils.pathutils	**0.707**
flutils.setuputils.cmd	0.500
flutils.strutils	0.500
httpie.cli.dicts	0.500
httpie.cli.exceptions	0.500
httpie.config	**0.883**
httpie.models	0.394
httpie.output.formatters.colors	**1.00**
httpie.output.formatters.headers	0.500
httpie.output.formatters.json	0.500
httpie.output.processing	**1.00**
httpie.output.streams	**0.016 7**
httpie.plugins.base	**0.867**
httpie.plugins.manager	0.550
httpie.sessions	**0.737**
httpie.ssl	0.500
httpie.status	0.500
isort.comments	0.500

continued on next page ...

... continued

Module Name	Effect Size
isort.exceptions	0.500
isort.utils	0.500
mimesis.builtins.base	0.500
mimesis.builtins.da	0.500
mimesis.builtins.de	0.500
mimesis.builtins.it	0.500
mimesis.builtins.nl	0.500
mimesis.builtins.pt_br	0.500
mimesis.builtins.uk	0.500
mimesis.decorators	0.500
mimesis.exceptions	0.500
mimesis.providers.choice	**0.352**
mimesis.providers.clothing	0.500
mimesis.providers.code	0.500
mimesis.providers.development	0.500
mimesis.providers.hardware	0.500
mimesis.providers.numbers	0.500
mimesis.providers.science	0.500
mimesis.providers.transport	0.500
mimesis.providers.units	0.500
mimesis.shortcuts	0.500
pdir._internal_utils	**0.612**
pdir.attr_category	**0.891**
pdir.color	0.500
pdir.configuration	0.500
pdir.format	**0.967**
py_backwards.conf	0.500
py_backwards.files	**0.**
py_backwards.transformers.base	**1.00**
py_backwards.transformers.class_without_bases	**0.883**
py_backwards.transformers.dict_unpacking	**1.00**
py_backwards.transformers.formatted_values	**0.950**
py_backwards.transformers.functions_annotations	0.500
py_backwards.transformers.import_pathlib	0.500
py_backwards.transformers.metaclass	**0.917**
py_backwards.transformers.python2_future	0.500
py_backwards.transformers.return_from_generator	**0.951**
py_backwards.transformers.starred_unpacking	**0.813**
py_backwards.transformers.string_types	0.500
py_backwards.transformers.variables_annotations	0.500
py_backwards.transformers.yield_from	**1.00**
py_backwards.types	0.500
py_backwards.utils.helpers	0.630
py_backwards.utils.snippet	**1.00**
pymonet.box	0.500
pymonet.immutable_list	0.500
pymonet.lazy	**0.383**
pymonet.maybe	0.500
pymonet.monad_try	0.500
pymonet.semigroups	0.500
pymonet.task	0.500
pymonet.validation	0.500
pypara.accounting.generic	0.500
pypara.accounting.journaling	0.467

continued on next page ...

... continued

Module Name	Effect Size
pypara.commons.errors	0.500
pypara.commons.numbers	0.500
pypara.commons.others	0.500
pypara.commons.zeitgeist	0.500
pypara.monetary	0.532
pytutils.debug	0.500
pytutils.excs	0.500
pytutils.files	0.500
pytutils.lazy.lazy_import	0.473
pytutils.meth	0.500
pytutils.path	0.500
pytutils.pretty	0.500
pytutils.props	0.500
pytutils.python	0.500
pytutils.pythree	0.500
pytutils.rand	0.500
sanic.base	0.500
sanic.config	0.500
sanic.cookies	0.408
sanic.handlers	0.500
sanic.headers	**0.676**
sanic.helpers	0.500
sanic.mixins.listeners	0.500
sanic.mixins.middleware	0.500
sanic.mixins.routes	**0.709**
sanic.mixins.signals	0.500
sanic.models.futures	0.500
sanic.models.protocol_types	0.500
sanic.views	0.583
string_utils.errors	0.500
string_utils.manipulation	**0.906**
string_utils.validation	0.447
sty.lib	0.500
sty.register	0.500
sty.renderfunc	0.500
thonny.languages	0.500
thonny.plugins.pgzero_frontend	0.500
thonny.roughparse	0.462
thonny.terminal	0.500
thonny.token_utils	0.500
typesystem.tokenize.positional_validation	0.500
typesystem.tokenize.tokenize_yaml	**0.583**
typesystem.unique	0.500