

PYSEAT: Python Symbolic Execution Engine and Automated Testing Tool



Juan Manuel Copia
Director: Pablo Ponzio

Diciembre 2020

Universidad Nacional de Río Cuarto.
Facultad de Ciencias Exactas, Físico-Químicas y Naturales.
Departamento de Computación

Abstract

Plenty of efforts are made to improve the quality of software. Most of them are thorough manual software testing, which involves human resources and time. Testing automation can improve software quality while reducing the costs of software testing. In this work, we developed a novel testing automation tool called PySEAT. It automatically generates test cases for Python programs that use complex data structures. PySEAT only takes as input the classes implementing the structure and the class invariant and automatically generates a test suite. PySEAT implements three different test input generation strategies. We based our work on an existing symbolic execution tool called PEF: Python Error Finder. PEF is a verification tool based on symbolic execution that doesn't work well with Python programs that use complex, heap-allocated data structures. Therefore, we reused part of the symbolic types and the SMT solver interface of PEF and built upon it a whole new symbolic execution engine, capable of handling complex data structures through the use of a technique called lazy initialization. We also added a new module that writes test cases from the results of the program's exploration. We assessed our tool and its three input generation strategies over several study cases taken from open source repositories (e.g. Binary search trees, AVLs), and measured the quality of the generated test suites. In all these study cases, PySEAT's best test input generation strategy produces test suites achieving high mutation score and branch coverage. We also have found a bug using our tool in an open source AVL implementation. Our evaluation also shows that PySEAT is capable of generating tests for covering code of data structures' implementations that PEF could not cover.

Acknowledgments

To my family and friends for their love and unconditional support.

To Pablo Ponzio for his great guidance and dedication.

Table of Contents

1. Introduction	1
2. Background.....	6
2.1. Software Testing.....	6
2.1.1. Black-box Testing	7
2.1.2. White-box Testing	8
2.2. Class Invariants	8
2.3. Traditional Symbolic Execution.....	9
2.3.1. Test Generation Example Using Symbolic Execution	11
2.4. Generalized Symbolic Execution	15
2.4.1. Generalized Symbolic Execution Example	18
3. The Technique	21
3.1. A Simple and Representative Example	23
3.2. Black-Box Strategy	24
3.2.1. Generalized Symbolic Execution of Class Invariant and Concretization.....	24
3.2.2. Traditional Symbolic Execution of the SUT	25
3.2.3. Discussion.....	26
3.3. White-Box Strategy Using Fully Conservative Invariants	26
3.3.1. Generalized Symbolic Execution of the SUT.....	28
3.3.2. Filtering and Building.....	32
3.3.3. Discussion.....	34
3.4. Exhaustive White-box	35
3.4.1. Generalized Symbolic Execution of the Class Invariant	36

3.4.2. Traditional Symbolic Execution of the SUT	37
3.4.3. Discussion.....	39
4. Implementation.....	41
4.1. Requirements	43
4.2. Parsing and Instrumentation	44
4.3. Symbolic Types	47
4.4. Branching	49
4.4.1. Conditional Branching.....	49
4.4.2. Lazy Initialization Branching	51
4.5. The Generalized Symbolic Execution Engine.....	52
4.5.1. The Explore Method.....	56
4.5.2. The Results	58
4.6. Test Generation.....	60
5. Experimental Evaluation	62
5.1. Circular Doubly Linked List	63
5.1.1. Discussion.....	65
5.2. Binary search tree	66
5.2.1. Discussion.....	68
5.3. AVL.....	69
5.3.1. Discussion.....	71
5.3.2. A Bug Found	71
5.4. PEF vs PySEAT	73
5.4.1. Singly Linked List	74
5.4.2. Binary Search Tree	75
6. Related work.....	77

7_Conclusion.....	79
7.1. Future work	81
References	84
A_Usage of PySEAT.....	87
A.1. Installation	87
A.2. Parameters	88
A.3. Execution	89
A.3.1. Measurement of branch coverage and mutation score	90

Chapter 1

Introduction

In the last decades, there has been an enormous growth in the software industry; software systems are everywhere. As in many other disciplines, software engineering is prone to human errors. The consequences of these failures range from crashes in uncritical systems, like video games, to big catastrophes in critical systems such as the Ariane's 5 explosion [20] or NASA's Mars climate orbiter [21]. Although many software engineering techniques are used to increase the quality of systems, testing is the primary way industry evaluates software during development [14].

Manual software testing is very costly; takes time and human resources. The automation of this practice can reduce the cost of software development while increasing the quality of software systems. In the last decades, the software engineering community has extensively explored automatic test generation techniques, originating several approaches and implementations. To mention some of the most important, we can highlight Randoop [9], based on random generation, Evosuite [10], based on genetic algorithms, and PEX [11], based on symbolic execution, among others.

Symbolic execution allows us to explore the paths of a program using symbolic values as input [8]. It executes the code under test and constructs formulas holding the constraints that inputs must satisfy to cover paths of the program (path conditions). We can use a constraint solver [12] to solve each of these path conditions, producing an input that exercises that path. Symbolic execution based approaches are popular among

researchers because, usually, they produce test suites that achieve high code coverage of the system under test (SUT.) Nevertheless, modern constraint solvers can only handle predicates over arrays and basic data types, as integers, floats or strings. To reason about dynamically allocated structures, we require other approaches [1,2,13]. The existing symbolic execution tools for the Python language, PEF [3] and PyExZ3 [6], don't implement these techniques. Therefore, they cannot achieve high code coverage when dealing with Python programs using complex dynamic data structures. In this work, we present a novel test automation tool for Python programs capable of handling these kinds of constructs, called PySEAT.

Python is one of the most popular programming languages nowadays, thus having a test automation tool with these characteristics can be of great importance. Moreover, Python is a commonly used programming language in computer science education, thus this tool can be effective to teach automated, symbolic execution-based test generation techniques.

We based our implementation on a tool called Python Error Finder (PEF) [3]. PEF implements an algorithm that uses the class constructor of user-defined classes to create inputs for the program under test. It first creates the input instance with the class constructor. For each of its reference fields, it calls the class constructor to initialize them. The algorithm continually keeps calling class constructors for each reference field of new user-defined instances until it reaches a predefined maximum number of objects the input can have. For instance, for a singly linked list, it successively calls the class constructor of the node class to initialize the "next" field of new nodes until it reaches the limit in the amount of nodes. This approach works well with data structures that do not require aliasing constraints, like acyclic singly linked lists, but it cannot handle more complex user-defined structures (i.e. data structures containing aliasing constraints), like double

linked lists or any tree with parent references. For instance, it cannot create valid instances of a circular linked list because it cannot satisfy the constraint that the last element of the list must point to the first element of the list.

Unlike other symbolic execution tools (e.g. Java PathFinder [15], DART [16]), PEF is built as an external library that runs together with the target program without changing the interpreter or instrumenting the code under test. Although PySEAT is also built as an external library, it adds code instrumentation to support complex heap allocated structures. We built PySEAT on top of PEF's symbolic types and SMT solver interface. We also added a test generation module to automatically create test suites for the SUTs.

The hearth of our tool is a novel generalized symbolic execution engine that allows the analysis of Python programs using complex data (i.e. data structures that can contain aliasing). Generalized symbolic execution combines traditional symbolic execution with a technique called lazy initialization to support dynamically allocated complex constructs [1].

There are two approaches for test input generation using generalized symbolic execution: the white-box and black-box strategies [2]. The black-box approach generates test inputs by performing generalized symbolic execution over the class invariant (it doesn't consider the SUT's code). The white-box approach generates test inputs by performing generalized symbolic execution over the SUT and uses the class invariant to assure the validity of those inputs. Here we implement three approaches, the two mentioned, and another strategy called "Exhaustive white-box" combining features of both.

PySEAT takes as input the system under test and automatically generates a test suite to check the correctness of its implementation. It requires the class under test (CUT) to have a class invariant implemented as a method of the API, usually called `repOK`. PySEAT uses class invariants to build inputs that are valid regarding the specification of the SUT. Test cases can also exploit class invariants to expose defects by checking if it holds after the program's execution.

The original white-box approach [2] requires that the user implements a particular type of class invariants to work, conservative class invariants. To the best of our knowledge, there is no sound algorithm to automate the creation of such class invariants from any given class invariant. Therefore, to automate this process and prevent the user from implementing a conservative class invariant, we used what we call fully conservative class invariants (FCI). This allowed the tool to work with any Python class invariant, but costs in terms of efficiency of using FCI were very high. We tested our tool using the three strategies over several study cases taken from open source repositories [19]. On all of them, the exhaustive white-box and the white-box strategies generate test suites that achieve higher mutation scores and branch coverage than the black-box. In several cases, the white-box approach using FCI exceeded timeout, and thus, it could not exercise an important part of the paths of the SUT. Nevertheless, when white-box could finish its execution, it achieved the same branch coverage and mutation score as the Exhaustive white-box, but with a smaller test suite.

We also have found an undetected bug in an open-source implementation of an AVL tree using PySEAT. PEF and PyExZ3 cannot reveal this bug because they cannot create AVL instances to exercise the paths of the SUT. Besides, to reveal this bug, we need test inputs (AVL instances) with at least 4 nodes. Therefore, during the experimental evaluation, the white-box approach cannot find the bug because the execution time

exceeds the timeout. The bug can only be found using PySEAT's best strategy (The exhaustive white-box) and the black-box strategy. We also compare the performance and capabilities between PySEAT and PEF over two study cases, an acyclic singly linked list, and a binary search tree with parent references. The results show that PEF cannot generate valid inputs to exercise the binary search tree implementation, whereas PySEAT was able to cover all the branches of the code. Also, PySEAT was faster than PEF in the singly linked list implementation.

Chapter 2 explains the background concepts to understand this thesis. Chapter 3 describes our work: the different input generation strategies, its advantages and disadvantages. Chapter 4 describes the implementation details of PySEAT. Chapter 5 shows the experimental evaluation of the three test generation strategies for data structures from open source repositories [19], using branch coverage and mutation score to evaluate the generated test suites. Also, the last section of this chapter presents the comparison between PySEAT and PEF. In chapter 6 we present some related works. Finally, in chapter 7 we discuss conclusions and future work.

Chapter 2

Background

2.1. Software Testing

Software testing is an approach to assess the functional correctness of a software system. It assesses the degree in which the system complies with its specifications, and its primary aim is to find bugs [14]. A test case is the atomic unit that exercises a specific system behavior. It comprises three parts: First, preparing suitable inputs for the system under test (SUT). Second, executing the system under test with those inputs. And third, verifying it produces the expected results. A test case fails if the result differs from the expected, usually exposing a defect in the SUT. The mechanisms used to perform this comparison are called "test oracles". We call test suites to sets of test cases. Figure 1 illustrates these concepts.

Branch coverage and mutation score are two common metrics to measure the quality of a test suite. Every control flow structure (e.g. if statements, loops) in the code can have two outcomes: it evaluates to either true or false. We call these outcomes "branches" and we say that a branch is covered if it was executed. For instance, if an execution of an if statement evaluated to true, we say that the true branch was covered. The branch coverage measures the percentage of executed branches in the program [14]. Mutation score measures the percentage of mutants killed by the test suite. A Mutant is an alteration in the target's code. We can think of it as a seeded defect in the system under test. A test case kills a mutant if it detects that the behavior of the mutant differs from the original version [14]. In other words, a test case kills a mutant if it passes for the original

program but fails for the mutant. Therefore, the measurement of mutation score comprises generating mutants of the program, executing the test suite over them, and calculating the percentage of killed mutants. In the practice, software testers use automated mutation and code coverage tools to measure these metrics.

Figure 1

Pseudo-code example of a test suite.

```
sum_two(a: int, b: int) {  
    return a + b  
}  
  
test_case1() {  
    // input configuration  
    x = 5  
    y = 10  
    // Program execution  
    result = sum_two(x, y)  
    // Test oracle  
    assert(result == 15)  
}  
  
test_case2() {  
    // input configuration  
    x = -3  
    y = 5  
    // Program execution  
    result = sum_two(x, y)  
    // Test oracle  
    assert(result == 2)  
}
```

Note. On the left, the program under test. On the right, a test suite with two test cases.

2.1.1. Black-box Testing

Black-box testing is a method that examines the functionality of the system under test without considering its internal structure. When creating test suites, the awareness is on what the system does, but not how it does it. Testers do not inspect the code of the program (sometimes because they don't have access to). Instead, they base on the system's requirements and specifications to write test cases [14].

2.1.2. White-box Testing

Unlike Black-box Testing, White-box testing specially considers the internal structure of the software system under test. It focuses specifically on how the system does its task. Testers inspect the program and derive tests from its source code internals, specifically including branches, individual conditions, and statements [14].

2.2. Class Invariants

Object-oriented programming languages represent user-defined data structures as classes of objects. We call objects to instances of a certain class. Complex data structures often involve a set of constraints that all their instances must satisfy. For instance, the constraint of traditional singly linked lists is the absence of cycles. This set of constraints compose the class invariant. Instances violating any constraint are invalid and for this reason, data structure's implementations mustn't allow creating invalid instances through the use of their API methods.

Figure 2

Class invariant of an acyclic singly linked list.

```
class SinglyLinkedList {
    Node head

    bool repok() {
        if (this.head == null)
            return true
        visited = set()
        visited.add(this.head)
        current = this.head
        while (current.next != null) {
            if (current.next already visited)
                return false
            current = current.next
        }
        return true
    }
}
```

To be correct, an implementation must meet the following requirements:

- **All instances created by the constructor must satisfy the class invariant.**
- **Each method of the class must preserve the class invariant:** Starting from an instance satisfying the class invariant, the execution of any method of the class produces an instance for which the class invariant holds.

Class invariants are an excellent resource to verify the behavior of implementations, and we can implement them as a class-specific method, usually called "repOK". Its task is to analyze the object's state and return true if it holds the class invariant, or false otherwise. Figure 2 shows a repOK for a singly linked list data structure, checking the acyclicity constraint.

2.3. Traditional Symbolic Execution

Symbolic execution is a technique that allows us to execute a program on a set of classes of inputs. Inputs exercising the same path of the program belong to the same class of input. Therefore, each symbolic execution result may be equivalent to a larger number of normal test cases [8]. To represent these classes of inputs, symbolic execution uses symbolic values as input (instead of concrete ones). Symbolic values are symbolic expressions which represent the range of potential values that a concrete execution could produce [4]. Initially, these expressions represent any value in the type's domain, but when the program performs an operation on the symbolic value, the expression might change. For instance, if we perform symbolic execution of the code of figure 3, the initial state of the variables x and y are the expressions **X** and **Y** respectively, which represent any integer value. We will denote symbolic expressions with uppercase bold letters and variables with lowercase italic letters. Therefore, the initial state of the variables is:

$$x: \mathbf{X}, \quad y: \mathbf{Y}$$

And after the execution of line 1, the expression of the variables x changes to:

$$x: \mathbf{Y * 2}, \quad y: \mathbf{Y}$$

Figure 3

Example of control flow

```

foo(x: int, y: int) {
1:   x = y * 2
2:   if (x > 10)
3:     return x
4:   else
5:     return y
}
```

Besides the symbolic values, another component that characterizes the state of symbolically executed programs is the path condition. The path condition is the set of constraints the input must satisfy to exercise that path of the program. During symbolic execution, the program's control flow statements (e.g. if statements, loops) add constraints to the path condition only if a symbolic value is involved. For instance, the if statement of line 2 will only add constraints to the path condition involving the symbolic value $\mathbf{Y * 2}$ (pointed by the x variable), because the control flow statement does not involve the symbolic value \mathbf{X} .

We also call the conditions of control flow statements "branch points", because they produce a ramification of the program execution. The branch point of line 2 has two ramifications: assuming " $\mathbf{Y * 2} > 10$ " or assuming " $\neg \mathbf{Y * 2} > 10$ ". As symbolic execution must explore all feasible branches, the algorithm can choose one arbitrarily and add the corresponding constraint to the path condition. Once the execution of a path ends, the algorithm backtracks and chooses another branch to explore. The algorithm repeats this process until it executes all feasible paths. Loops can create infinite branching, so the algorithm must bound the amount of nested ramifications it can perform.

The set of all the constraints collected along the execution of a program path comprises the path condition. At the beginning of the symbolic execution, it is empty. The path condition characterizes the inputs that causes the program to follow that path. We can use a constraint solver [12] to solve those constraints and generate the inputs and the test case to cover that path. Constraint solvers, also called SMT solvers, can reason about expressions (constraints) involving different theories like integers, booleans, arrays, etc. They provide models (solutions) to these constraints. Models are mappings of values to each variable in the expressions that make all the constraints satisfiable.

The symbolic execution of the example executes two paths, one in which the path condition is " $Y * 2 > 10$ " and other In which it is " $\neg Y * 2 > 10$ ". By solving these constraints we get two solutions, for example: $[Y=6, X=0]$ and $[Y=1, X=0]$. Executing the program with these inputs, we cover all its paths.

2.3.1. Test Generation Example Using Symbolic Execution

Consider the Pseudo-code of the function in figure 4. The program has a defect on line 8, represented by the false assertion. When it is executed, the program crashes. Here, the goal of any tester or any automated testing tool is to create a test case that executes the defect. To do that, we need to find out the input that causes the program to follow that path. By performing symbolic execution, we can get this information.

Figure 4

Pseudo-code of a function with a defect.

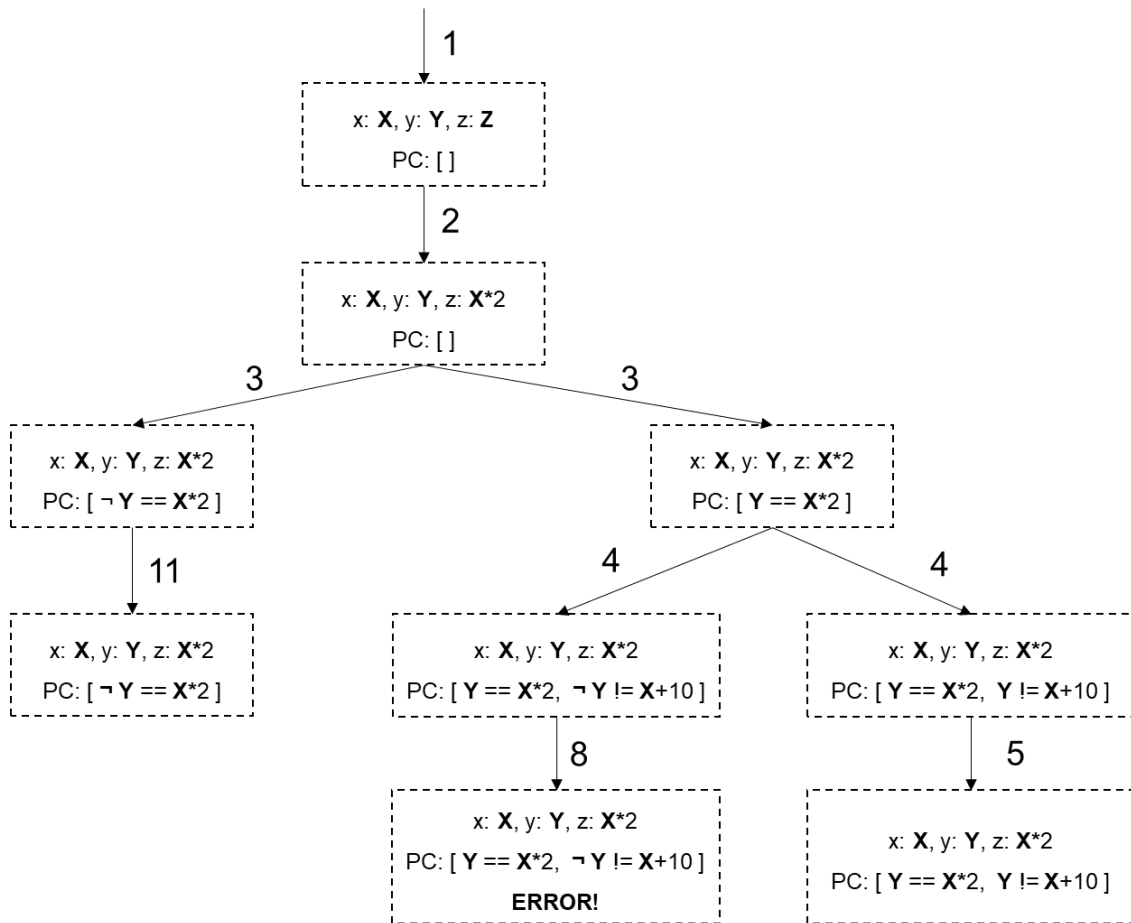
```
1: int func(x: int, y: int) {
2:     int z = x * 2
3:     if (y == z) {
4:         if (y != x + 10) {
5:             return z
6:         }
7:         else {
8:             assert False
9:         }
10:    }
11:    return z
12: }
```

Note. The false assertion on line 8 represents a defect in the code.

If we consider all the states in the symbolic execution process of the function, we get the tree of Figure 5. We refer to this tree as the “symbolic execution tree”. Each node represents the state of the execution, showing the state of each variable and the path condition (PC). The process begins with an empty path condition and the symbolic expressions representing any value of the domain (in our case, any integer). The arcs represent execution of sentences that lead to each state. Bifurcations in the tree correspond to conditions in the control flow statements in the program (sentences 3 and 4); the algorithm chooses a path and continues its execution. Note that after the choice, it adds the corresponding constraint to the path condition. When the execution of that path ends, the algorithm backtracks and chooses the other alternative. It repeats this process until it executes all paths. Eventually, it executes the path with the defect.

Figure 5

Symbolic execution tree of funcion on figure 3



Note. Arrows with number n represents the execution of the sentence number n In the function “func” of figure 4. The squares represent the state of the symbolic execution at each point.

The symbolic execution finds three paths in the program. These paths correspond to executing the lines: [1,2,3,11], [1,2,3,4,8] (this is the one executing the defect), and [1,2,3,4,5].

Table 1

Results of symbolic execution example.

	Input	Return	Path condition	Solution
1	$x = X, y = Y$	ERROR	$[Y == X * 2, \neg (Y != X + 10)]$	$y = 20, x = 10$
2	$x = X, y = Y$	$X * 2$	$[Y == X * 2, Y != X + 10]$	$x = 2, y = 4, Ret = 4$
3	$x = X, y = Y$	$X * 2$	$[\neg (Y == X * 2)]$	$x = 3, y = 2, Ret = 6$

By solving the constraints of each final path condition, we get the inputs to execute each of the respective program paths. Table 1 describes the three executions results, along with a solution that we can get from a constraint solver. With this information, we can create the three test cases in figure 6, each of them exploring exactly one path from the program under test. The test suite is minimal in this sense and achieves 100% of branch coverage of the program.

Figure 6

Test suite result of the symbolic execution of function of figure 4.

```

test1() {
    // ERROR!
    func(10, 20)
}

test2() {
    assert(func(2, 4) == 4)
}

test3() {
    assert(func(3, 2) == 6)
}

```

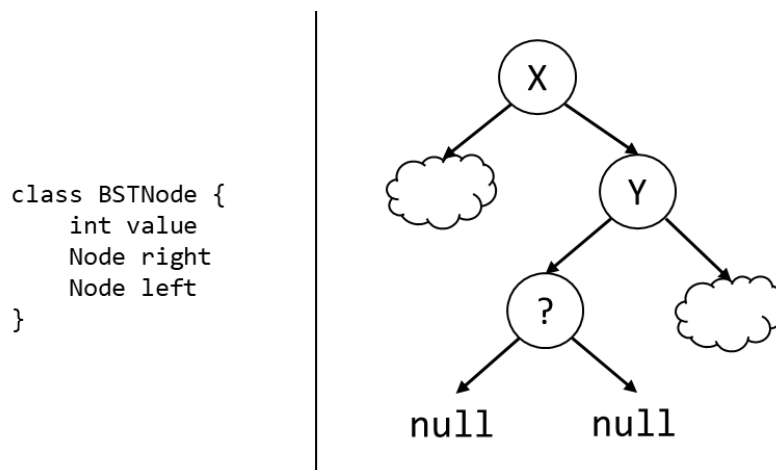
2.4. Generalized Symbolic Execution

The original symbolic execution approach [8] cannot handle dynamically allocated structures on the heap. Among other approaches, Generalized symbolic execution (GSE) [1] extends traditional symbolic execution to address this problem.

As traditional symbolic execution, GSE uses symbolic inputs instead of concrete ones. Particularly, it represents heap allocated structures by "partially symbolic structures". We can define partially symbolic structures as instances that might contain uninitialized fields. These fields represent undefined parts of the instance that could take any value of its type domain. Figure 7 illustrates a partially symbolic instance of a set of nodes forming a binary search tree. Clouds represent uninitialized reference fields, the "?" sign represents uninitialized primitive fields, and uppercase letters in nodes represent symbolic integer values.

Figure 7

Example of a partially symbolic binary search tree.



Note. On the left, the node of a binary search tree. On the right, an illustration of a partially symbolic instance of a BSTNode.

The core idea of GSE is to start the execution of the program with a partially symbolic instance with all its fields uninitialized. During the execution of the program, the algorithm initializes the fields on demand. i.e. when they are first accessed. The initialization depends on the field. On one hand, the algorithm initializes primitive fields to its symbolic counterpart. On the other, it initializes reference fields (structures) non-deterministically to one of the following options:

- Null.
- A new, partially symbolic instance (with uninitialized fields) of the field's type.
- An instance created during a prior initialization of a field of the same type.

Any of these alternatives can produce instances violating the method precondition or the class invariant. When this happens, the algorithm backtracks, discarding the instance and choosing another initialization option.

This initialization procedure is called lazy initialization [1]. Figure 8 depicts a pseudo-code of the algorithm.

Figure 8

Lazy Initialization algorithm

```

if ( f is uninitialized ) {
  if ( f is reference field of type T ) {
    nondeterministically initialize f to
      1. null
      2. a new object of class T (with uninitialized field values)
      3. an object created during a prior initialization of a field of type T
    if ( method precondition is violated )
      backtrack();
  }
  if ( f is primitive (or string) field )
    initialize f to a new symbolic value of appropriate type
}

```

Note. Pseudo-code of the lazy initialization algorithm. Source: [1].

While traditional symbolic execution branches only on control flow statements involving primitive symbolic values (we also call this conditional branching), Generalized symbolic execution also branches on initializations of reference fields (lazy initialization branching). When the execution of a path ends, it backtracks and explores another branch. It repeats this process until it executes all feasible branches. However, like symbolic execution, cycles can create infinite branching. Therefore, it also needs a bound on the amount of nested branching it can perform. A common practice is to set a maximum number of objects that the input can contain.

From each path in the generalized symbolic execution of the program, we get the path condition (with the same characteristics as in symbolic execution) and instances of the input structures. With this information we could create a test case to follow each path, but we must have several considerations regarding to the final state of the structures:

- **Destructive updates:** If the method changes the input structure (e.g. the method `add` of a linked list), the final state of that instance differs from the original input. Thus, to handle programs that perform destructive updates, the algorithm builds mappings between objects with uninitialized fields and objects that are created when those fields are initialized; these mappings are used to construct input structures [1].
- Once the algorithm recovers the original input, it might still have uninitialized fields. To make these structures useful for testing, we need to initialize those fields in a way that the class invariant holds. To achieve this, one can run GSE on the class invariant with the partially symbolic instance as input. This process initializes uninitialized fields and we get a complete valid instance.

2.4.1. Generalized Symbolic Execution Example

Pseudo-code of figure 9 checks if a singly linked list is sorted in strictly ascending order. We're going to execute the process of generalized symbolic execution step-by-step on this example. The input of this method is the implicit input "this", i.e. an instance of a singly linked list. We are assuming acyclicity as class invariant. To keep the example small, we are going to limit the number of objects the input can contain to two. To follow the state of the execution, we are going to use the generalized symbolic execution tree of this method (figure 10). The caption describes the meaning of the symbols present in the chart. It's important to mention that here, we traverse the tree in an order of our convenience. But keep in mind that in practice the algorithm might use a different traversing order, like depth first search (executing a whole path and then backtracking to explore other options).

Figure 9

Pseudo-code method that checks if a singly linked list is sorted

```

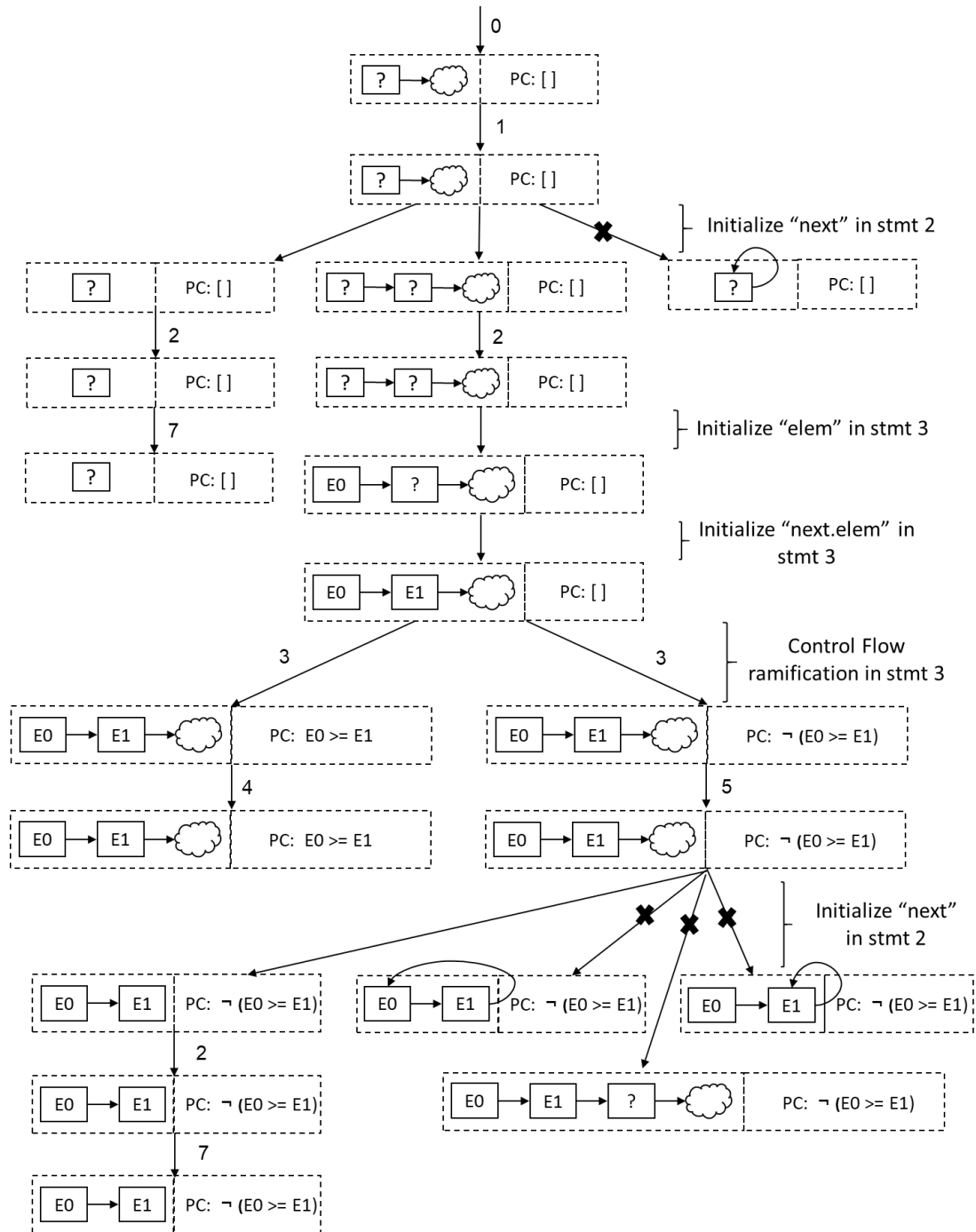
class Node {
    int elem
    Node next

    bool is_sorted() {
1:     Node current = this
2:     while (current.next) {
3:         if (current.elem >= current.next.elem)
4:             return false
5:         current = current.next
6:     }
7:     return true
    }
}

```

Figure 10

Generalized symbolic execution tree of "is_sorted" method



Note. The tree comprises the states of the generalized symbolic execution of the "is_sorted" method of figure 9. The states are the nodes of the tree, represented by dashed line squares. Each state has two parts, on the left, the instance configuration and on the right, the path condition. Clouds on the instance represent an uninitialized reference field "next", the "?" sign represents an uninitialized integer field "elem", the symbols "E0" and "E1" are symbolic integers (initializations of the field "elem"), and solid line squares represent node instances. Arrows with number k are

execution of the sentence k of the method. Arrows without a number represent lazy initializations. Arrows marked with an "X" represent pruned paths.

The process starts with an instance of the linked list with uninitialized fields as input. Line 2 produces the first access to the reference field "next", and the algorithm non-deterministically initializes the field to null, a new instance with uninitialized fields, and to an existing instance of the same type (itself being the only option). The last option violates the class invariant, so it is discarded. The null initialization makes the program follow the false branch and return true on line 7 (the list only has one element, so it is sorted). The initialization to a new instance makes the program follow the true branch. Before executing line 3, the algorithm initialized the "elem" fields of both nodes. The execution of line seven corresponds to a control flow branching. For each branch, the algorithm adds the corresponding constraint to the path condition. The constraint for the true branch " $E0 \geq E1$ " means the list is unsorted, so the algorithm returns false on sentence 4. The false branch continues its execution until, again, before the evaluation of the while condition, the algorithm performs an initialization step. This time, the options are four because there are two existing node instances. The algorithm discards three of them; two due to a violation of the class invariant and one because it exceeds the limit of structures. The only valid initialization makes the program follow the false branch and return true on sentence 7 (the constraint is " $E0 < E1$ ", thus the list is sorted).

To sum up, GSE explores three valid paths on the "is_sorted" method. From each of them, we get the input structure and the constraints over its primitive values (path condition). As in traditional symbolic execution, we can use a constraint solver to get the concrete inputs and generate the test cases that exercise those paths.

Chapter 3

The Technique

In this work, we have developed a new tool that automatically generates test cases for Python programs that handle complex dynamically allocated structures, PySEAT. The tool takes as input an implementation of a complex data structure (e.g. Binary Search Trees, AVLs), and generates a test suite. It relies on the use of class invariants, implemented as a method of the class under test, usually referred to as "repOK". Although it can be easily extended, the current implementation of PySEAT only supports the testing of methods that take as input the implicit self and parameters of the types integer and bool. Our primary focus is the support of complex user-defined structures and the implementation and analysis of three input generation strategies.

Recall from the previous chapters that we can use symbolic execution with lazy initialization to achieve this purpose. We developed a whole new generalized symbolic execution engine capable of handling complex user-defined constructs. To achieve this, we adapt some components from an existing symbolic execution python tool called PEF. Particularly we reuse parts of its SMT interface and symbolic (primitive) types.

One of the most important parts of test generation is the construction of inputs. The inputs determine the effectiveness of the test suite in achieving high code coverage and in finding faults. We can take several approaches of test input generation using the generalized symbolic execution procedure. In [2] they describe the black-box and white-box strategies. During the development of the tool, our first attempt was to implement the

white-box approach. The major problem was that it relies on a specific type of class invariants, conservative invariants. As we wanted our tool to work with any class invariant, we had to make some changes in the strategy. Although the primary aim wasn't to achieve high efficiency, we found that these "changes" generated important efficiency issues. On our second attempt, we tried the black-box approach; it had better performance, but the quality of the test suites generated in terms of branch coverage wasn't good enough. Therefore, we come up with a novel approach to address these problems. It combines the black box and white box approach described in [2] taking advantage of the best qualities from both. We called it the "Exhaustive White-box" approach, and PySEAT uses it by default.

This chapter is organized in the following way. In section 3.1, we present an example that we will use to compare the different input generation strategies. Section 3.2 describes the black-box strategy. Section 3.3 describes the white-box strategy and the problems and disadvantages we have found. Later, on 3.4 we will explain our novel approach, the Exhaustive White-box, how it addresses the problems of white-box and black-box, its advantages and disadvantages. Finally, in 3.5 we describe the implementation details of the entire process. From parsing and instrumentation, to the module that transforms the results of the execution in test cases.

3.1. A Simple and Representative Example

Throughout the next sections, we will compare the three input generation strategies using an example that illustrates the primary characteristics of each approach.

Figure 11

Acyclic singly linked list implementation in Python

```

class LinkedList:
    head: "Node"

    def __init__(self):
        self.head = None

    def is_sorted(self):
1:     current = self.head
2:     while current and current.next:
3:         if current.elem >= current.next.elem:
4:             return False
5:         current = current.next
6:     return True

    def repok(self):
    # acyclic
    visited = set()
    current = self.head
    while current:
        if current in visited:
            return False
        visited.add(current)
        current = current.next
    return True

class Node:
    elem: int
    next: "Node"

    def __init__(self, elem: int):
        self.elem = elem
        self.next = None

```

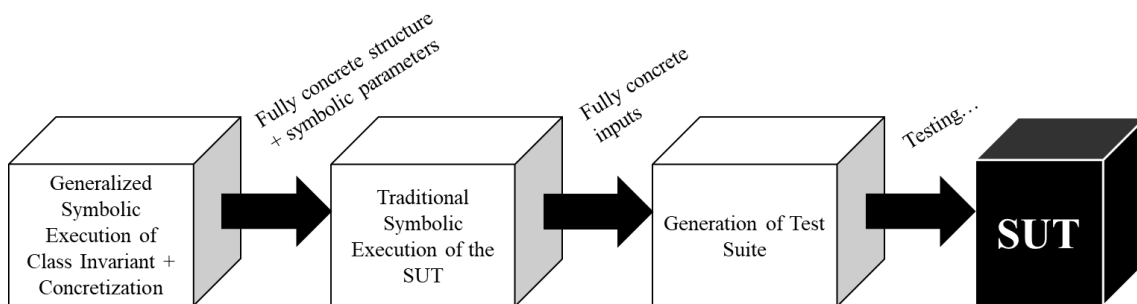
Our target is the "is_sorted" method from the acyclic singly linked list of figure 11. It checks whether the integers in the list are sorted in strictly ascending order. We will limit the number of nodes of the linked list to 4. That will be enough to appreciate the difference between strategies. Note that the repOK method implements the class invariant. It checks the acyclicity of the list.

3.2. Black-Box Strategy

The key point of this input generation strategy is that it constructs the inputs for testing the SUT by performing generalized symbolic execution of the class invariant. It generates inputs without concern for the SUT's internal structure. We can divide test generation using this approach in the three stages depicted in figure 12.

Figure 12

Black-box test generation steps



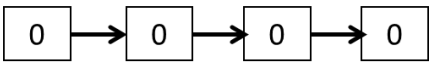
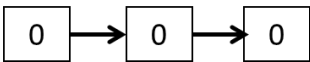
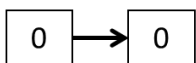

3.2.1. Generalized Symbolic Execution of Class Invariant and Concretization

We perform an exploration of the class invariant (“repOK” method) using GSE. Each path of the class invariant has two potential outcomes: it returns true, or it returns false. We ignore paths returning false because it means the instance it creates is spurious. From the exploration of each of the paths satisfying the class invariant (paths returning true), we get two things: First, a partially symbolic structure. And second, a path condition containing a set of constraints that the primitive fields of the structure must satisfy. For instance, for each path of the repOK method in figure 11, we get a partially symbolic instance of a list, and an empty path condition because the invariant doesn’t require any

constraints over the primitive values. Thus, by solving path condition constraints using a constraint solver and concretizing the structure (replacing symbolic values with concrete ones) with the solver model, we got a fully concrete input that holds the class invariant. Figure 13 shows the instances we get for the linked list example. It's important to mention that if the class invariant doesn't impose restrictions over some fields of the instance, they might still be uninitialized. This is not a problem because it means they could take any value, thus we can initialize them to null (in case of reference fields) or to the default value (in case of primitive fields) during concretization. From a structural standpoint (without considering primitive values), we get all possible valid structures up to a bound.

Figure 13

Black-box input generation for acyclic singly linked list

Test Inputs	Expected Result
	False
	False
	False
	True
<empty-list>	True

Note. Test inputs generated by the Black-box strategy for the method "is_sorted" from a singly linked list implementation. The bound in the number of nodes is four.

3.2.2. Traditional Symbolic Execution of the SUT

We use each concrete structure generated in the previous step as input to perform symbolic execution of the SUT, instantiating other parameters (if exist) with the

corresponding symbolic type. Finally, we ask the SMT solver for a model to concretize the symbolic parameters. For our example, this stage works as a normal execution because in the method "is_sorted" there aren't other parameters. From this stage, we get the full concrete inputs, so we can create the test cases for the SUT.

3.2.3. Discussion

For the linked list example, we get five test cases, one for each instance in figure 13. These inputs are not very useful; as we are testing a method that checks order, it would be better to have more input lists with unique integer values arranged in distinct orders. It doesn't happen because the class invariant doesn't require any constraints over the integer values of the list, i.e. during GSE of repOK, the path condition ends up empty, thus the concretization initializes them to zero. Besides, the traditional symbolic execution stage explores the target with an input structure containing only concrete values, so the constraints of the path condition don't affect this input either. Therefore, when using the black-box approach, the input structure will be the same regardless of the SUT. This usually causes the test suite to miss important parts of the SUT's code.

3.3. White-Box Strategy Using Fully Conservative Invariants

The key feature of this approach is that it tailors the inputs to cover the paths of the SUT. As a result, we get a small set of inputs that achieve high coverage of the program under test. In the black-box approach, we performed GSE over the class invariant to eliminate spurious instances. Here, we first perform GSE over the SUT and use the class invariant to avoid initializations on the input's fields that cause it to be invalid. The original white-box approach [1,2] use only conservative class invariants. To allow it to

work with any class invariant, we use instead fully conservative class invariants (FCI). This decision brought several efficiency issues. Most of them emerge when dealing with more complex structures than linked lists. For this reason, during the following sections, besides the linked list example of section 3.1, we will use the binary search tree example of figure 14 to explain those problems. Figure 15 shows the three stages of this approach: Generalized symbolic execution of the SUT, filtering and building, and test case generation.

Figure 14

Fragment of a binary search tree Python implementation

```

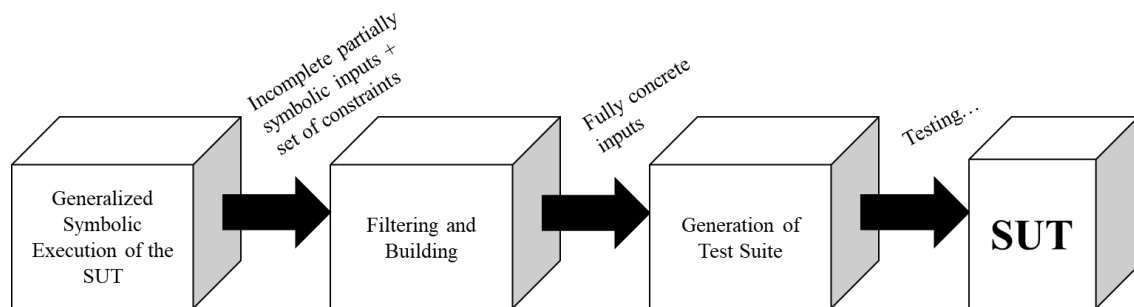
class Node:
    def __init__(self, value: int):
        self.value = value
        self.left = None
        self.right = None
        self.parent = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, value: int):
1:   if self.root == None:
2:       self.root = Node(value)
3:   else:
4:       self._insert(value, self.root)

    def _insert(self, value, curr):
5:   if value < curr.value:
6:       if curr.left == None:
7:           curr.left = Node(value)
8:           curr.left.parent = curr
9:       else:
10:          self._insert(value, curr.left)
11:  elif value > curr.value:
12:      if curr.right == None:
13:          curr.right = Node(value)
14:          curr.right.parent = curr
15:      else:
16:          self._insert(value, curr.right)
17:  else:
18:      print("Value already in tree!")

```

Figure 15*White-box test generation steps*

3.3.1. Generalized Symbolic Execution of the SUT.

This stage comprises performing generalized symbolic execution directly on the program under test. As we saw in chapter 2, when performing a lazy initialization step, we need to ensure that the initialization is valid, i.e., that it satisfies the class invariant. But since we are dealing with structures with uninitialized fields, the checking of the class invariant must be conservative. That is, to use the original white-box approach [2], the user has to implement a `repOK` that returns false only if the initialized fields violate a constraint of the class invariant. To the best of our knowledge, there is no sound algorithm to automate the creation of such `repOK` from an existing one. As we didn't want the user to deal with the complexities of coding a conservative invariant, we used instead what we called a "fully conservative invariant" (FCI). The difference is that during FCI execution over the partially symbolic instance, it returns true as soon as it accesses an uninitialized field. This is a simpler process than the original conservative invariants, and any `repOK` can be checked automatically in a fully conservative way, but it brings several other issues. Figure 16 shows on the left the original class invariant, and on the right, a python code representing the behavior of that class invariant when executed in a fully conservative way.

Figure 16

Concrete class invariant and fully conservative class invariant (FCI) for a binary search tree

```

def repok(self):
    if not self.root:
        return True
    if not (self.is_acyclic()):
        return False
    if not (self.is_ordered()):
        return False
    return True

def is_acyclic(self):
    visited = set()
    visited.add(self.root)
    worklist = []
    worklist.append(self.root)
    while worklist:
        curr = worklist.pop(0)
        if curr.left:
            if not do_add(visited, curr.left):
                return False
            worklist.append(curr.left)
        if curr.right:
            if not do_add(visited, curr.right):
                return False
            worklist.append(curr.right)
    return True

def full_conservative_repok(self):
1:  if is_not_initialized(self.root):
2:      return True
3:  if not self.root:
4:      return True
5:  if not (self.is_acyclic()):
6:      return False
7:  if not (self.is_ordered()):
8:      return False
9:  return True

def is_acyclic(self):
10: visited = set()
11: visited.add(self.root)
12: worklist = []
13: worklist.append(self.root)
14: while worklist:
15:     curr = worklist.pop(0)
16:     if is_not_initialized(curr.left):
17:         return True
18:     if curr.left:
19:         if not do_add(visited, curr.left):
20:             return False
21:         worklist.append(curr.left)
22:     if is_not_initialized(curr.right):
23:         return True
24:     if curr.right:
25:         if not do_add(visited, curr.right):
26:             return False
27:         worklist.append(curr.right)
28: return True

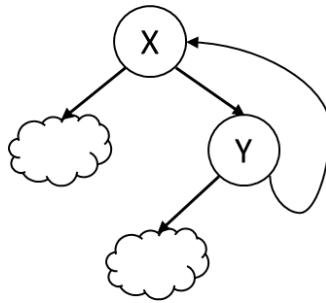
```

Note. On the left, the concrete class invariant. On the right, a representation of the behavior that the class invariant on the left has when executed in a fully conservative way. The code lines in bold characterizes the unconventional behavior.

One problem of FCIs is that they cannot detect many spurious instances. Frequently, when a lazy initialization step creates an invalid instance, the fully conservative repOK cannot reach the wrong initialization. For instance, consider the binary search tree implementation of figure 14 and the FCI of Figure 16. If we execute the fully conservative class invariant over the spurious instance of figure 17, the repOK finds first the uninitialized left field of the root (line 16), and returns true (line 17). Thus, it cannot reach the invalid initialization (the cycle of the right child of the root), and the execution continues with a spurious structure that passes unnoticed.

Figure 17

Spurious instance of a binary search tree



These spurious instances produce further problems. As the methods expect to receive valid instances as input, spurious instances can generate infinite loops and stack overflows. For example, during the exploration of the "insert" method of figure 14, lazy initialization steps can generate the spurious instance in figure 17, producing a stack overflow in the recursive call in line 16 (figure 14). The trace that generates the stack overflow is: [1,3,4,5,11,12,15,16]. To solve these issues, we set a limit in the amount of times the execution can call a getter method of a field with no field being initialized.

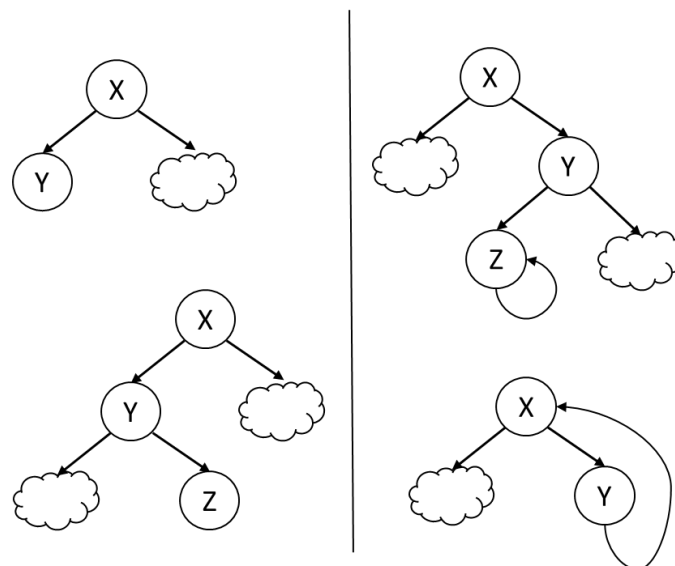
We also have to consider the cases when the method under test performs destructive updates over the input. In these cases, the structure we get at the end of the exploration is not the original input. To take care of this, during the execution of each path, we keep two structures: The one that is being used and changed by the method, and another in which the only modifications are the applications of initialization steps, i.e. after each successful initialization on the destructively updated structure, our algorithm performs the same initialization in the "control" structure. Thus, this second structure represents the original input, and we use it as input for test cases.

Like in the black-box input generation, the result of the generalized execution is a partially symbolic instance of the data structure, and a set of constraints over its primitive values. But since here we are not executing the class invariant, the instance we get might

still have uninitialized fields. This is because the target method may not explore all parts of an input structure. And thus, some fields are not initialized. Therefore we can't concretize the structure directly as in the previous approach. Like the linked list example is simple, we end up only with valid list instances with all its fields initialized. But when dealing with more complex structures, like the BST of figure 14, these situations are very frequent. For instance, Figure 18 illustrates the result of the exploration of some paths of the insert method of the BST. It contains valid and invalid instances. To create valid test cases, we need to eliminate the invalids and build the valid ones. By "build" we mean to initialize uninitialized fields in a way that the structure is valid and "concretize" the structure (replace symbolic values for concrete ones). We perform these actions in the next stage.

Figure 18

Partially symbolic instances of a BST tree



Note. The Clouds represent uninitialized reference fields. The instances of the right are invalid because they contain initialized cyclic references.

3.3.2. *Filtering and Building*

To create valid test cases, we need to use valid structures as input. Thus first, we need to remove the spurious structures from the previous stage. And second, we need to build the valid structures. The only asset we have to decide whether a structure is valid is the class invariant. Therefore, to detect spurious structures and to build valid ones, we perform GSE on the class invariant. i.e. we explore the repOK method (with a limit in the number of structures it can add) with the partially symbolic instance as input and its constraints as the initial state of the path condition. This process will initialize uninitialized fields until one of the next things happens:

1. **The repOK finds a path that returns true:** It means that the structure was not spurious and that it found a valid initialization. The instance is now “built” and the algorithm will use it as a test input.
2. **The repOK finishes and all its paths returned false:** It could mean that the structure was invalid (it had invalid initialized fields). Remember that the class invariant will only instantiate uninitialized fields. Or it could also mean that the structure was valid, but the bound was not enough to build it. In both cases, the algorithm discards the structure.

The algorithm does this process for each instance it creates in the previous stage. Note that case 2 is a heavy process computationally, because it performs a complete exploration of the class invariant. As the number of infeasible structures produced most times is very large and detecting them is very heavy computationally, the white-box strategy with fully conservative invariants is very inefficient.

Figure 19

Pseudo-code that filters spurious structures and completes valid ones

```

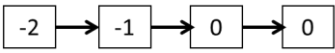
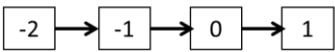
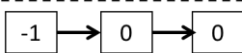
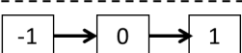



filter_and_build(partially_symbolic_instances: list) -> list:
    concrete_instances = []
    for instance in partially_symbolic_instances:
        concrete_ins = build_and_concretize(instance, number_nodes)
        if concrete_ins:
            concrete_instances.append(concrete_ins)
    return concrete_instances

```

Finally, if it was possible to build the instance, we concretize it by asking the SMT solver to solve the constraints. Figure 19 shows the pseudo-code that describes the behavior of the entire stage. Figure 20 shows the inputs that this stage generates for the example of section 3.1.

Figure 20

White-box input generation for acyclic singly linked list

Test Inputs	Expected Result
	False
	True
	False
	True
	False
	True
	True
<empty-list>	True

Note. Test inputs generated by the White-box strategy for the method "is_sorted" of the singly linked list implementation. The bound in the number of nodes is four.

3.3.3. Discussion

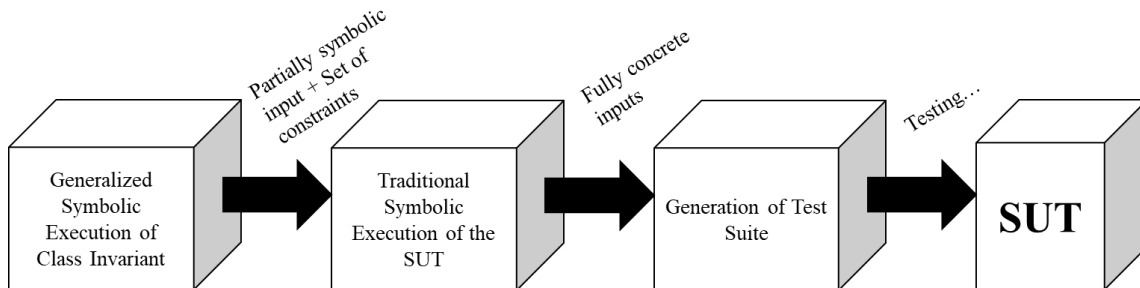
Each of the inputs in figure 20 covers exactly one path of the generalized symbolic execution tree, without repetition. This is because the algorithm executes the SUT, adding constraints to the inputs that cause the program to follow different paths. Thus, the test suite we get is usually small and achieves high branch coverage for the program under test.

To make this approach work with any class invariant, we use fully conservative invariants (FCI) instead of conservative invariants as the original approach [1,2]. We make this decision because we didn't want the user to deal with the complexities of coding a conservative class invariant, and because, to the best of our knowledge, there is no sound algorithm to convert a concrete class invariant into a conservative one. We can run any class invariant in a full conservative way, but this procedure cannot detect all spurious instances during lazy initialization steps, and thus, these invalid instances provoke several problems: infinite loops during GSE of the SUT, unnecessary computations and the need to perform GSE over the class invariant with the instance as input to erase invalid constructs before test generation. Solving all these problems, especially the last one, makes the implemented strategy very inefficient.

3.4. Exhaustive White-box

Black-box approach generates test inputs that are not "guided" by the structure of the code in the SUT. Thus, in many cases, it misses covering important parts of the code. On the other hand, the white-box approach excels in this aspect. Because it constructs the inputs specifically to achieve the coverage of the SUT. Nevertheless, the approach we took to automate it lacks efficiency; when dealing with more complex data structures, like AVLs, the time to generate test cases is enormous even for little scopes. Thus, it cannot cover the parts of the code that need larger inputs. To sort these obstacles, we have developed a new strategy that takes the best out of both approaches. We called it Exhaustive White-box.

Our aim is to perform a white-box input generation to achieve high code coverage of the target, but more efficiently. The core idea is to avoid the root cause of most of the downsides of the white-box strategy: the use of fully conservative invariants. The function of a conservative invariant is to eliminate invalid initializations during GSE of the SUT. To get rid of it, we need a different way of removing spurious constructs. We accomplish this purpose by using a process similar to the black-box technique to solve structural constraints before exploring the target method. In this way, there is no need to perform GSE on the SUT because we already have the input, and thus, we only perform traditional symbolic execution of the program under test.

Figure 21*Exhaustive White-box test generation procedure*

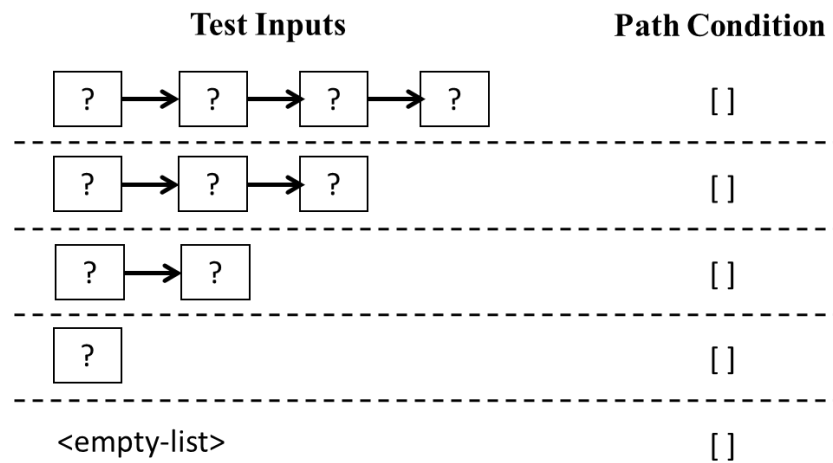
This strategy works in three stages: Generalized symbolic execution of the class invariant, traditional symbolic execution of the method under test, and finally, test case generation.

3.4.1. Generalized Symbolic Execution of the Class Invariant

This step is like the first step of the black-box technique, with the difference that at the end of the class invariant's exploration, it doesn't concretize the instances. The black-box method solves the constraints over primitive inputs and gets a valid concrete input structure. Here, we don't concretize the structures because those constraints contain valid information that we can take advantage of; they describe the classes of values that make the instance valid. Therefore, we use those partially symbolic instances and the constraints over its primitive values as input to perform traditional symbolic execution on the SUT. Figure 22 shows the results applied to the linked list example. The integer elements of the list are not initialized because the repOK method doesn't access them. The path condition is always empty for the same reason.

Figure 22

Results of this stage applied to the example in section 3.1



3.4.2. Traditional Symbolic Execution of the SUT

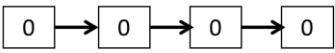
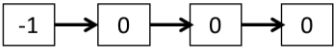
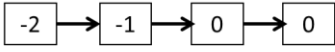
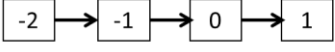
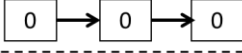


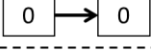


For each pair (instance, constraints) from the previous stage, this step uses the symbolic instances as input to perform traditional symbolic execution of the SUT, using its constraints as the initial state of the path condition. During the symbolic execution of each pair, the target method will add new constraints to the path condition. These new constraints shape the primitive values of the input to explore different branches of the SUT. A nice consequence of using the constraints from the previous stage as the start state of path condition is that during method exploration, the algorithm discards the execution of paths that violate the existing constraints, producing further pruning.

From each instance from the previous step, this stage will probably create several inputs, varying its primitive values but maintaining the structural shape. Finally, we have to concretize the primitive fields using the model provided by the SMT Solver. Figure 23 shows the results when testing the linked list example. Note that from a structural standpoint (i.e. taking into account reference fields only), it is exhaustive (there is an

empty list, lists with only one node, with two nodes, and so on). We can also observe that for each structural possibility it covers all feasible primitive constraints from the code. For instance, for lists containing two nodes, it covers both values of the constraint in line 3 of figure 11: $First\ int \geq Second\ int$, and $\neg First\ int \geq Second\ int$. After this stage, the only thing left is to create the test cases.

Figure 23

Exhaustive white-box input generation for acyclic singly linked list

Test Inputs	Expected Result
	False
	False
	False
	True
	False
	False
	True
	False
	True
	True
<empty-list>	True

Note. Test inputs generated by the Exhaustive white-box strategy for the method "is_sorted" of the singly linked list implementation. The bound in the number of nodes is four.

3.4.3. Discussion

The core characteristic of this last strategy is to first create the inputs by running GSE over the class invariant, like the black-box approach, but keeping those inputs symbolics. This process discards the spurious instances generated during GSE and initializes all the fields that determine the validity of the structure. Consequently, during the symbolic execution of the SUT, it avoids lazy initialization steps and thus, the use of Fully conservative class invariants, that were the root cause of inefficiency in our white-box approach implementation. It's also important to mention that it can work with any class invariant, and not only with conservative ones. Another consequence is that it avoids special handling of methods performing destructive updates because we already know what the initial input is.

We say that this method is a combination of the white-box and the black-box approaches for two reasons. On one hand, from a structural standpoint, it behaves as the black-box strategy because it creates the structure of the inputs (configuration of the reference fields) guided by the class invariant instead of the SUTs code. This leads to an exhaustive generation of inputs in terms of their structure, i.e. the inputs it creates are all the valid configurations of reference fields. On the other hand, from the primitive values standpoint, it behaves like a white-box strategy; the value of primitive fields depends strongly on the SUT.

As we can see from our experimental section, both the Exhaustive white-box and the white-box using fully conservative invariants get the same branch coverage and mutation scores for scopes where both can reach in the defined timeout. A key advantage of the white-box strategy is that it generates smaller test suites, but because of its inefficiency, it usually exceeds the time limit even for small scopes, generating smaller

inputs that cannot cover important parts of the SUT's code. On the flip side, the Exhaustive white box creates plenty of test cases that exercise the same path, making the test suite enormous.

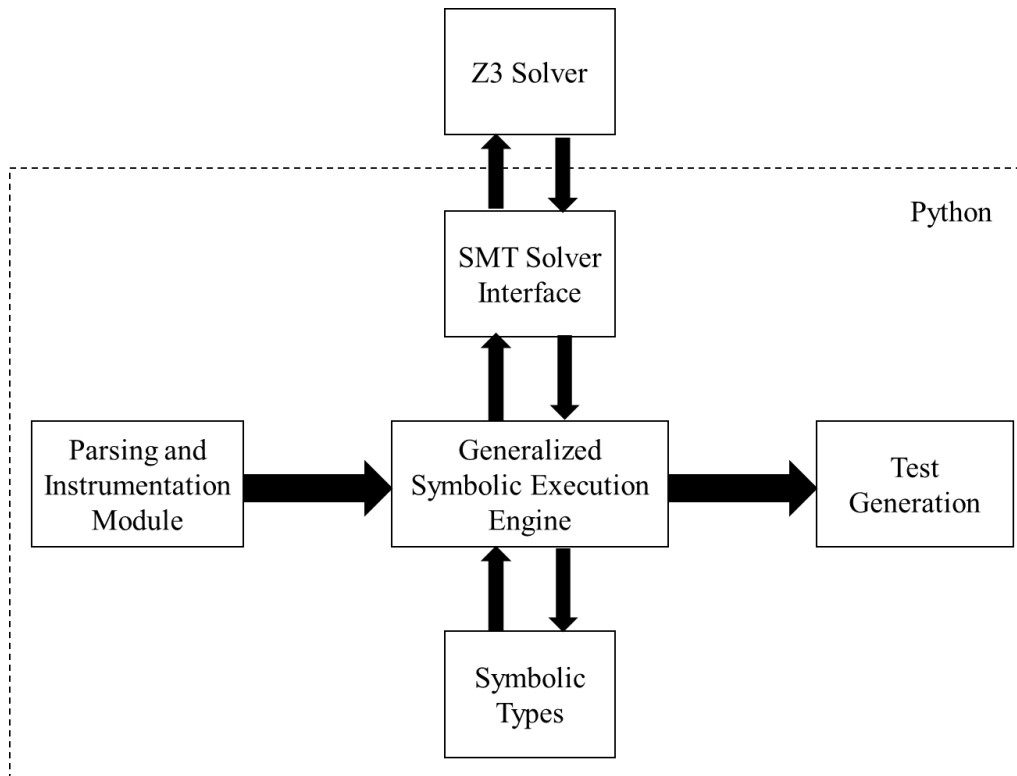
Chapter 4

Implementation

In this chapter, we present the implementation of PySEAT, its principal components, the role that each of them has and the interactions between each other. Figure 24 depicts the architecture of PySEAT. We implement symbolic execution using the Z3 SMT Solver. We also reuse and adapt the Symbolic types and the SMT interface from the PEF tool [3].

Figure 24

PySEAT's architecture



The major components of PySEAT are:

- **Parsing and Instrumentation module:** To perform generalized symbolic execution, the engine requires an instrumentation of the SUT's code and also information about the data types of its objects and methods. The function of this component is to fulfill that requirement by parsing the types and instrumenting the code.
- **Generalized Symbolic Execution Engine:** It is the core component of the entire system. Its function is to perform the symbolic exploration of the instrumented SUT. To do so, it coordinates the interactions between Symbolic Types and the SMT Solver. For this reason, it is highly attached to them. Finally, it gathers the exploration results for the Test Generation module.
- **SMT Solver:** It's key to the symbolic execution process; it decides the satisfiability of path conditions and provides the required models to concretize the input and output structures. The SMT interface it's an API to allow the communication between the engine and Z3.
- **Symbolic Types:** They replace the concrete primitive types during the symbolic execution. Symbolic types emulate the behavior of the primitive ones but symbolically, by holding symbolic expressions that characterize the value of the symbolic instance. These symbolic expressions are formulas from the Z3 solver describing operations between Z3 variables. Initially, the formula of a symbolic instance is a reference to a Z3 variable. That is, symbolic values of each symbolic type instance are represented by a variable from the Z3 solver. The Z3 variables can represent single values (e.g. a symbolic value \mathbf{X}) or operations between single values (e.g. $\mathbf{X} + \mathbf{Y}$). Operations over symbolic instances generate the formula describing the same operation between the Z3 solver variables.

- **Test Generation:** It takes the results of the exploration and generates the code of the test cases.

4.1. Requirements

One objective we had in mind during the development was to minimize the effort that the user must perform to use the tool. The goal was to automate trivial tasks as much as possible. Therefore, we reduced the requirements needed by PySEAT to just two:

- **Implementing a repOK method on the Class Under Test:** This method implements the class invariant and is necessary to discard invalid instances of the class during symbolic execution. This avoids the generation of test cases with spurious instances.
- **Adding of type annotations:** Python is a language with dynamic typing; types of variables and parameters are not present in the code. But since Python 3, the language supports type annotations of parameters and fields. The engine requires knowing the types to initialize objects with the appropriate symbolic type. Therefore, PySEAT expects the user to annotate the following elements on the SUT through Python Annotations:
 - **The constructor of every "relevant" class:** The parameters of the constructor must be annotated. The self annotation can be omitted because it is always inferred. We say a class is "relevant" when it is the type of a parameter of some method under test, or when it is the type of a field in the CUT or in any other relevant class.
 - **The formal parameters of every method under test:** Every method under test must have its formal parameters annotated. Again, self annotation can be omitted.

To sum up, the user has to annotate the types of the relevant classes (if they are not annotated yet) and implement the class invariant as a method of the class called "repOK". With this information, PySEAT can generate the test suite. Figure 25 shows an example of the type annotations over a doubly linked list.

Figure 25

Example of types annotations for a doubly linked list

```
class DoublyLinkedList:
    # Instance attributes must be annotated.
    head: "Node"
    tail: "Node"

    # Init params if exists must be also annotated.
    def __init__(self):
        self.head = None
        self.tail = None

    # Methods under test must be annotated.
    def insert_at_front(self, value: int):
        node = Node(value)
        if self.is_empty():
            self.head = node
            self.tail = node
        else:
            node.next = self.head
            self.head.prev = node
            self.head = node

class Node:
    # Instance attributes must be annotated.
    data: int
    next: "Node"
    prev: "Node"

    # Init params if exists must be also annotated.
    def __init__(self, data: int):
        self.data = data
        self.next = None
        self.prev = None
```

4.2. Parsing and Instrumentation

To allow the generalized symbolic execution engine to work, we need to parse the type annotations and instrument the SUT's code. To be clearer about how we implement this process, we are going to use the concrete example of figure 25. The parsing is done

automatically by the interpreter, so we will not enter into the details of this process. On the other side, understanding the instrumentation is key to understanding how the symbolic execution engine works. The instrumentation comprises the addition of class attributes and instance attributes to each relevant class in the SUT.

Class attributes are variables that belong to the class rather than a particular instance. All instances of a class share the same class attributes. We add the following class attributes:

1. **The `_engine` field:** It is a reference to the symbolic execution engine. It allows the class to call methods of the engine instance. The function of this reference is strongly correlated to the addition of Python properties to manage access to each field of the class (see item 3 below).
2. **The `_vector` field:** It is a list that during GSE, it will contain all the class instances created during lazy initialization steps.
3. **Python properties to manage access to each field of the structure:** Properties are a feature of Python that allows managing access over fields of the instances [17]. More precisely, they allow specifying what to do when the value of the field is required (acting as a custom getter method) and what to do when the value of the field is changed (acting as a custom setter method). Thus, through python properties, we add our custom getters and setter methods for each field:
 - **The getter method:** it derives the getting method to the method "`_get_attr`" from the engine.
 - **The setter method:** It derives the responsibility of setting the value to the method "`_set_attr`" from the engine.

Through these python properties, we achieve to "hook" every access to the fields of an instance. The engine has full responsibility for getting and setting the corresponding values.

Instance attributes are variables that belong only to each particular instance, i.e. they are only accessible in the scope of the instance they belong. The algorithm adds these variables to each instance after they are created. They are:

1. **An initialized mark for each field of the class:** This mark is true when the field is initialized, and false otherwise.
2. **An id field:** A unique number that identifies the instance.

Returning to our example, figure 26 shows the instrumented doubly linked list. It's important to remark that PySEAT never changes the source code of the target on the file. It's only an illustrative example. Because of the flexibility and capabilities of Python, PySEAT does the instrumentation process dynamically rather than statically. We achieve the desired result by adding the mentioned elements in runtime instead of changing the source code before program execution. This process is transparent to the user. The module **instrumentation.py** adds the instrumentation class attributes. The engine adds the rest of the instrumentation instance attributes when it creates partially symbolic instances during the generalized symbolic execution.

Figure 26*Example of Instrumentation for a Doubly linked list*

```

class DoublyLinkedList:
    # Instance attributes annotations.
    head: "Node"
    tail: "Node"

    _engine = None # reference to engine.
    _vector = [] # vector of instances.

    # Property for the field "head".
    @property
    def head(self):
        return self._engine._get_attr(self, "head")

    @head.setter
    def head(self, value):
        return self._engine._set_attr(self, "head", value)

    # Property for the field "tail".
    @property
    def tail(self):
        return self._engine._get_attr(self, "tail")

    @tail.setter
    def tail(self, value):
        return self._engine._set_attr(self, "tail", value)

    def __init__(self):
        self.head = None
        self.tail = None
        self._is_init_head = False # initialized mark for head.
        self._is_init_tail = False # initialized mark for tail.
        self._id = -1 # instance id.

```

Note. This is only an illustrative example of the instrumentation for the doubly linked list of figure 25. PySEAT instruments the classes dynamically.

4.3. Symbolic Types

As we said before, we took the symbolic types implementation from PEF [3]. PEF takes advantage of a key Python feature to implement symbolic execution. In Python, and in many other languages, all data are objects. Thus, all operations over objects (including primitive types) are translated into method calls of these objects. For instance, during the execution of a Python program, to evaluate the expression $x + y$, the Python interpreter calls `x.__add__(y)`. Moreover, Python is a dynamically typed language. These features allow us to implement symbolic execution by implementing the symbolic types and using

them instead of the primitive ones to execute the target program. Symbolic types must reimplement all the operations of their primitive counterpart, but symbolically. Instead of concrete values, they hold symbolic expressions which describe the operations applied between symbolic values. They create these symbolic expressions using the constraint solver (in our case, Z3). For instance, a symbolic integer must implement, among others, the method `__add__`. Thus when symbolically executing the same expression $x + y$, as x and y are symbolic integers, the Python interpreter calls `x.__add__(y)` that belongs to the symbolic integer instance `x`. The `__add__` method will create and return a new symbolic integer instance holding the Z3 expression (or formula) $X + Y$, where X and Y are variables from the Z3 SMT solver representing symbolic values.

In the same way, whenever the execution of a program needs the bool value of an object to continue its execution, the Python interpreter calls the method `__bool__` of that object, which returns true or false, depending on that instance. For example, in figure 27, when the execution reaches the if statement, the interpreter calls the `__bool__` method of the instance "myobj". This method will evaluate the instance and return its boolean value. If it returns true, the execution will execute the "then" block, otherwise it will execute the "else" block.

Figure 27

If statement in Python

```
if myobj:
    # some code
else:
    # other code
```

Note. When evaluating the if condition, the interpreter calls the method `__bool__` of "myobj" which decides its boolean value.

4.4. Branching

As we saw in section 2.4, there are two kinds of branching possibilities during generalized symbolic execution: Conditional branching and lazy initialization branching.

4.4.1. Conditional Branching

To perform symbolic execution, PySEAT replaces concrete arguments by their symbolic counterparts (e.g. `SymInt` for integers, `SymBool` for booleans). Therefore, we represent all boolean expressions involving symbolic objects by instances of `SymBool`.

Consider also the example of figure 28. When we execute this function symbolically, the arguments `x` and `y` are symbolic integers. When the execution reaches the "if" statement (line 1), Python interpreter resolves it in two steps:

1. It calls the corresponding function that implements the operator, in this case, the expression `x > y` is resolved by calling `x.__gt__(y)`. As it is a boolean expression, the `__gt__` method of the symbolic integer class returns a symbolic bool instance (`SymBool`) representing the expression `X > Y`.
2. Now that the expression is already resolved, we have to determine the truth value of the `SymBool` object for the program to continue its execution. Thus the interpreter calls the `__bool__` method from that symbolic bool instance to decide its value.

Figure 28

Python function that returns the maximum of two integers

```
def max(x: int, y: int):
1:   if x > y:
2:       return x
3:   else:
4:       return y
```


The if statement of figure 28 represents an example of conditional branching in a symbolically executed program. Conditional branching is generated whenever the interpreter reaches a condition in a control flow statement involving symbolic primitive values. In these cases, the algorithm of symbolic execution must consider the feasibility of both potential values of the symbolic expression (true and false). In these cases, the interpreter calls method `__bool__` of the symbolic expression. This method derives the responsibility of handling conditional branching to the engine. As shown in figure 29, the method "evaluate" is the handler of conditional branching from the engine.

Figure 29

__bool__ method from SymBool

```
def __bool__(self):
    formula = self.formula
    if isinstance(formula, bool):
        return formula
    return self.engine.evaluate(formula)
```

The engine handles conditional branching in a deterministic way; it always explores first the true branch and leaves the false branch for future exploration. It checks the feasibility of the decision with the current path condition using the constraint solver. It only proceeds with the execution of a branch if it is feasible. It is also optimized to check feasibility of the same conditional branch point only once; it remembers the answers of the constraint solver regarding the feasibility of a condition and a path condition.

4.4.2. Lazy Initialization Branching

As we saw in the instrumentation section, all accesses to the fields of a partially symbolic structure have a hook to the engine; during symbolic execution, the engine manages the initializations of all fields of the structure. Figure 30 shows a pseudocode of the engine's getter and setter methods.

Figure 30

Pseudocode of the methods from the engine that handle the access to the fields of structures

<pre>def _get_attr(field): if field_is_initialized(): return attr set_field_to_initialized() if field_is_user_defined_type(): new_value = lazy_initialization() else: # is a primitive field new_value = new_symbolic_value(type(field)) set_field(new_value) return new_value</pre>	<pre>def _set_attr(field, new_value): set_field_to_initialized() set_field(new_value) return new_value</pre>
---	--

When the value of a field is required, the **_get_attr** method is called, and it works as follows:

- If the field is already initialized, it returns the value of the attribute. Its type doesn't make a difference.
- If the field is uninitialized and it is of a primitive type, **_get_attr** marks it as initialized and sets its value to a symbolic instance of the corresponding type.
- If the field is uninitialized and it is of a reference type, **_get_attr** marks it as initialized and the method **lazy_initialization** performs a lazy initialization branching. As in conditional branching, the engine chooses initializations deterministically. It will

explore the first option and leave the others for future exploration. It follows the next order:

1. A new instance of the same type.
2. Null.
3. To each instance of the same type created during a prior initialization.

After the initialization, if the strategy in use is the white-box technique, the algorithm runs the class invariant in a full conservative way to check if the initialization makes the structure spurious. If it does, that branch is discarded. As we described before, it is possible that the FCI class invariant does not detect the invalid initialization because it first reaches an uninitialized field. If that is the case, the execution will be discarded during the process of filtering and building described in 3.3.2. In the black-box and exhaustive white-box input generation strategies, there is no need to check lazy initializations because in these cases we are performing symbolic execution over the class invariant, and thus, on invalid initializations the `repOk` method will return false and the algorithm will discard that input.

When a field is modified, the `_set_attr` method is called. This method only sets the field as initialized, initializes the field with the new value and returns that value.

4.5. The Generalized Symbolic Execution Engine

During this section, we will explain how PySEAT performs Generalized Symbolic Execution (GSE). Throughout the following paragraphs, we will refer to the term path. Keep in mind that we are referring to a path of the generalized symbolic execution tree, and not a path of the actual graph of the code. When we say a path is explored, it means it was executed using GSE.

The engine works by sequentially exploring one path of the program at a time. It replaces concrete values with primitive ones and starts the execution. As we explained in the previous sections, the control will be given back to the engine in every field access of the structure and in every branch point that occurs. PySEAT remembers the decisions taken in the different branch points along a path. It represents these decisions by the classes `ConditionalBranchPoint` and `LazyBranchPoint`. Both classes are implementations of the abstract class `BranchPoint` shown in figure 31. Instances of these classes hold the decision taken at that branch point. For instance, the conditional branch points hold whether the branch evaluated to true or false, while lazy branch points represent whether the lazy initialization step of some reference field was made to a new instance, null, or to an instance created in a prior step.

Figure 31

BranchPoint abstract class

```
class BranchPoint:
    """Branch point abstract class.
    """

    def get_branch(self):
        """Returns the current value of the branch point
        """
        raise NotImplementedError

    def all_branches_covered(self):
        """ Tells whether the branch point is fully covered or not.

        Returns:
            True if all possible branches of this branch point were
            covered, and if all the subsequent branch points
            generated by these decisions have also been fully
            covered, False otherwise.
        """
        raise NotImplementedError

    def advance_branch(self):
        """ Set the value of this branch point to the next one.
        """
        raise NotImplementedError
```

The core idea of this representation is that the algorithm can remember the path taken by stacking the BranchPoints generated during the exploration. A path in the symbolic execution tree is only characterized by the decisions taken in the branch points along that path. In that manner, by choosing always the same decisions, we traverse the same path again. And, by changing the decision on top of the stack, we traverse an unexplored path. This method leads to a bounded depth-first search exploration of the generalized symbolic execution tree.

For instance, consider the Python version of the "is_sorted" example in figure 32. There are only two sentences in the code that create branching. Sentence 2 corresponds to a lazy initialization of the reference field "next" (the field "next" of the "current" node is always first accessed at this line) and sentence 3 corresponds to a conditional branch point (the condition in the if statement involves the symbolic integer instances of "elem").

Figure 32

Python version of the "is_sorted" example

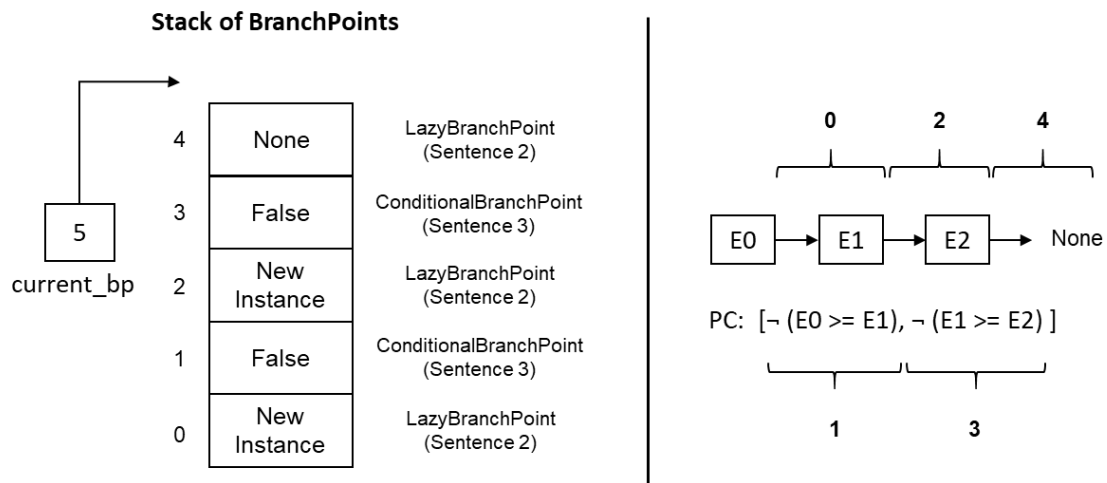
```
class Node:
    elem: int
    next: "Node"

    def __init__(self, elem: int):
        self.elem = elem
        self.next = None

    def is_sorted(self):
1:     current = self
2:     while current.next:
3:         if current.elem >= current.next.elem:
4:             return False
5:         current = current.next
6:     return True
```

Figure 33

Representation of the stack of branch points



Note. On the left, an example of a stack of branch points that describes the decisions taken along a path of the GSE of the **is_sorted** method, specifically the path: [1,2,3,5,2,3,5,2,6]. On the right, the input instance that is created because of those decisions. The numbers represent the branch point decision that generated that change.

The stack illustration of figure 33 belongs to a generalized symbolic execution of a path of the **is_sorted** method. Particularly, the path with the trace [1,2,3,5,2,3,5,2,6]. Note that the stack describes the initializations taken at each branch point. By taking the same decisions, we explore the same path, getting the input instance of the right side of the picture. If we change the decision on top of the stack (i.e., change the branch point at index 4 to an existing instance instead of null) and execute the program following those decisions, we explore another path of the generalized symbolic execution tree. The int variable **current_bp** is an index indicating the current branch point, in this case, it is pointing to an inexistent branch point (index 5).

Figure 34

Pseudocode of the method that manages the execution of the paths the target program

```

def explore(self, method):
1:  unexplored_paths = True
2:
3:  while unexplored_paths:
4:      self._reset_exploration()
5:
6:      result = self._explore_path(method)
7:      yield result
8:
9:      unexplored_paths = self._set_next_path()

```

4.5.1. The Explore Method

The exploration of the target program is managed by the "explore" method from the engine (figure 34). Initially, the stack of branch points and the path condition are empty, and **current_bp** is 0. The symbolic execution of a path of the program is triggered by the method **_explore_path** (line 6). Once the execution of a path starts, the control of the execution goes back and forth between the engine and the target program. The engine will intervene in each field access and in each branch point in the program. When the control is given back to the engine because of branching (for both conditional branching and lazy initialization branching), the engine handlers proceeds as follows:

- If there is a branch point in the stack for the **current_bp** index (i.e. if **current_bp** points to a valid index in the stack) it means that a prior exploration already determined the current path, and thus it takes that decision and advances the current branch point (**current_bp += 1**).
- If there is not a branch point in the stack for the current index, it means the engine is executing that branch point for the first time, and thus chooses an option, creating and adding the new branch point (describing the decision taken) at the

top of the stack. As the algorithm is deterministic, the first options will always be "true" for conditional branching and "new instance" for lazy initialization branching. After it adds the new branch point, it increases the current branch point (**current_bp += 1**).

Eventually, the execution of that program path will end and the explore method yields (line 7) the results generated during the execution of that path (these results comprise the inputs and outputs necessary to create a test case for that path). That leaves the engine with a stack of branch points characterizing the executed path.

Figure 35

Method that prepares the stack of branch points to the execution of the next path

```
def _set_next_path(self):
    if not self._branch_points:
        # All paths in symbolic execution tree were explored.
        return False

    last_bp = self._branch_points[-1] # get the top of the stack
    last_bp.advance_branch()          # change its value to the next one

    if last_bp.all_branches_covered(): # if it is the last branch option
        del self._branch_points[-1]   # remove the branch point from the stack
        return self._set_next_path()
    return True
```

Then, the method **_set_next_path** presented in figure 35 and called in the explore method at line 9, changes the decision at the top of the stack for the next one. For instance, if there is a LazyBranchPoint at the top of the stack describing the decision "New instance", it sets it to the next value, that is "none". If the branch point of the top of the stacks holds the last option of that branch point (e.g. a ConditionalBranchPoint instance holding the option "false"), the method pops the element in the top of the stack and calls itself recursively. This procedure causes the stack of branch points to describe a new path (a path never explored before). Finally, the method returns true if there are still branch

points on the stack, meaning there are still unexplored paths. It returns false if the stack of branch points is empty, meaning that there are no more paths to explore. If there are still paths to explore, the loops start again and the method **_reset_exploration** will set the **current_bp** to 0 and empty the path condition. This time, the stack of branch points already contains a set of branch points describing a path. Therefore, the method **_explore_path** will execute the program but this time following the decisions described by those branch points, which leads to an execution of an unexplored path. The entire process described continues until all feasible paths (bounded) are executed.

The engine bounds the exploration of paths with a limit in the maximum number of conditional branch points it can make, and with a limit in the maximum number of new instances it can create in reference field initializations. For instance, we can bound the "is_sorted" method to create instances with a maximum of five nodes and with a maximum of 10 conditional branch points allowed in the stack. These restrictions avoid infinite loops because the engine stops exploring the paths violating them.

4.5.2. The Results

When the execution of a path ends normally (it is not pruned by a violation of the bounds), the engine invokes the solver to get a model of the path condition's constraints. With the model, the engine concretizes the input and output of the execution. It also collects any exception raised and the final status of the execution. The following results are possible:

- PRUNED:
 - It exceeds the depth limit.
- OK:
 - It ends normally (doesn't throw exceptions).

- The structure that results from the execution satisfies the class invariant.
- FAIL:
 - It ends normally (doesn't throw exceptions).
 - The structure that results from the execution doesn't satisfy the class invariant.
- TIMEOUT:
 - It ends with a TimeoutException. The execution exceeded the time limit.
- EXCEPTION:
 - It ends with an exception different from TimeoutException.

As PySEAT already knows the status of the test, it will point out the expected result of each generated test case before it calls the test generation module. Figure 36 shows an example of the output of PySEAT during the exploration of a method. Finally, the algorithm passes the concretized elements to the test generation module.

Figure 36

PySEAT output during method exploration

```
Exploring method...  
  
#1 EXCEPTION  
#2 OK  
#3 TIMEOUT  
#4 OK  
#5 OK  
#6 OK  
#7 EXCEPTION  
#8 OK  
#9 FAIL  
#10 OK  
#11 FAIL
```

4.6. Test Generation

This module generates the test suite. Specifically, one test case for each path of the target method. Each test has the following parts:

1. **The input generation:** Comprises the Python code to generate the structure (the implicit input self).
2. **The call to the method under test.**
3. **A repOK invocation on the resulting instance:** To ensure the method under test leaves the input structure in a state that holds class invariant. The repOK is an oracle that allows PySEAT to find failures in the CUT.
4. **Regression assertions:** Assertions that check the outputs of the method call, including the final state of the input structure. They only check the current behavior of the target, so it doesn't mean it is necessarily the correct behavior. These assertions allow the test suite to work for regression testing.

Optionally, PySEAT can generate a test comment showing the input, the expected result and the expected output of the execution using the `__repr__` method of the classes.

PySEAT also generates special test cases when the execution of a path throws a `TimeoutException`. For these cases, it uses a feature of the test runner `pytest`, which sets a timeout for the test execution. When `pytest` runs this test, if the execution takes longer than the time limit, it reports it as a failure. Figure 37 shows a normal test case and figure 38 shows an example of a timeout test case.

These special test cases allow PySEAT to create test cases that reveal infinite loops. The time limit is the same used during method exploration and it is a parameter provided by the user. Although the tool uses the `pytest` test runner, any test runner can run the generated test cases.

Figure 37

A test case for the insert method of a Binary Search Tree generated by PySEAT

```

def test_insert2():
    '''
    Self:
        0
       / \
      -2  1

    Return:
        None
    End Self:
        0
       / \
      -2  1
       \
        -1
    '''
    1: # Input Creation
    2: bst0 = BST()
    3: node1 = Node(0)
    4: node1.data = 0
    5: node3 = Node(0)
    6: node3.data = 1
    7: node3.right = None
    8: node3.right = None
    9: node3.left = None
    10: node1.right = node3
    11: node2 = Node(0)
    12: node2.data = -2
    13: node2.right = None
    14: node2.left = None
    15: node1.left = node2
    16: bst0.root = node1
    17: # Method call
    18: returnv = bst0.insert(-1)
    19: # Repok check
    20: assert bst0.repok()
    21: # Regression assertions
    22: assert returnv is None
    23: assert bst0.root.data == 0
    24: assert bst0.root.right.data == 1
    25: assert bst0.root.right.right is None
    26: assert bst0.root.right.left is None
    27: assert bst0.root.left.data == -2
    28: assert bst0.root.left.left is None
    29: assert bst0.root.left.right.data == -1
    30: assert bst0.root.left.right.right is None
    31: assert bst0.root.left.right.left is None
    '''

```

1. Input generation

2. Method call

3. Repok assertion

4. Regression assertions

Figure 38

PySEAT test case that reveals an infinite loop in a method from a doubly linked list

```

@pytest.mark.timeout(2)
def test_insert_after17():
    '''
    Self:
        None <- 1 -> <- 0 -> None
    '''
    # Input Creation
    doublylinkedlist0 = DoublyLinkedList()
    node1 = Node(0)
    node1.data = 1
    node1.prev = None
    node2 = Node(0)
    node2.data = 0
    node2.next = None
    node2.prev = node1
    node1.next = node2
    doublylinkedlist0.head = node1
    doublylinkedlist0.tail = node2
    # Method call
    returnv = doublylinkedlist0.insert_after(0, 0)
    # Repok check
    assert doublylinkedlist0.repok()

```

Chapter 5

Experimental Evaluation

We assessed the three implemented strategies on three study cases taken from open source repositories: Circular doubly linked list, binary search trees, and AVLs [19]. We took the implementations from open-source repositories. We execute each strategy, varying the limit in the amount of nodes of the structure. The metrics we used to assess each strategy are: The amount of test it generates, the execution time of the test generation and the branch coverage and mutation score of the generated test suite. We only generate tests for the methods from each data structure relevant to its functionality. That is, we exclude from testing methods related to string representation (`__repr__`), methods related only to the class invariant (`repok`) and methods that the relevant methods depend on but they are not important by themselves (for instance, the rebalancing methods from an AVL were excluded from testing as they are tested indirectly by the methods `insert` and `delete`), although we do measure branch coverage and mutation score over this last set of methods.

We also compared the exhaustive white-box strategy with PEF. We contrast the performance of each tool over two data structures. A simple structure like a singly linked and a complex one like a binary search tree with parent references. We used the tools *coverage.py* and *mutmut* to measure branch coverage and mutation score, respectively.

Sections 5.1 to 5.3 shows the results for the different case studies. Specifically, in 5.3.4 we explain the bug our tool finds in the AVL implementation. Finally, in section 5.4 we compare our tool with PEF.

5.1. Circular Doubly Linked List

Figure 39

Circular doubly linked list definition

```

class CDLinkedList:
    head: "Node"

    def __init__(self):
        self.head = None

class Node:
    key: int
    next: "Node"
    prev: "Node"

    def __init__(self, key: int):
        self.key = key
        self.next = None
        self.prev = None

```

Figure 40 shows the definition of the classes that implements the data structure.

We generated tests for five methods:

- **insert_after(self, afterkey: int, key: int):** Inserts a new node in the list with the value "key" after a specific node with key "afterkey".
- **insert_before(self, beforekey: int, key: int):** Inserts a new node in the list with the value "key" before the first node with value "beforekey".
- **delete(self, deletekey: int):** Deletes the first node on the list with the value "delete key".
- **append(self, key: int):** Inserts a node with the value "key" at the end of the list.
- **prepend(self, key: int):** Inserts a node with the value "key" at the beginning of the list.

Table 2*Black-box statistics for Circular doubly linked list*

Node Limit	RepOK structures	Tests	Time (secs)	Branch Coverage	Mutation Score
1	2	13	0.5	70%	37.3%
2	3	21	1.1	88%	53.3%
3	4	29	1.8	88%	56.0%
4	5	37	2.6	88%	56.0%
5	6	45	3.6	88%	56.0%
6	7	53	5.0	88%	56.0%
7	8	60	6.2	88%	56.0%
8	9	67	7.8	88%	56.0%

In Table 2 we show the results for CDLL using the black-box strategy. The maximum branch coverage and mutation score this strategy achieves are 88% and 56% respectively. It reaches these bounds with a limit of only three nodes. Using over three nodes doesn't increase neither the branch coverage or the mutation score.

Table 3*White-box statistics for Circular doubly linked list*

Node Limit	Tests	Time (sec)	Branch Coverage	Mutation Score
1	13	0.8	70%	40.0%
2	24	2.9	100%	72.0%
3	33	6.0	100%	76.0%
4	41	11.9	100%	86.7%
5	49	18.3	100%	86.7%
6	57	27.5	100%	86.7%
7	65	43.4	100%	86.7%
8	73	71.5	100%	86.7%

In Table 3 we show the results for CDLL using the white-box strategy with FCI (fully conservative class invariants). It reaches the 100% branch coverage with a scope of only two nodes. The mutation score reaches its maximum (86.7%) when the node limit is four.

Table 4

Exhaustive White-box statistics for Circular doubly linked list

Node Limit	RepOK structures	Tests	Time (sec)	Branch Coverage	Mutation Score
1	2	13	0.5	70%	40%
2	3	24	1.1	100%	72%
3	4	38	2.0	100%	76%
4	5	55	3.2	100%	86.7%
5	6	75	4.6	100%	86.7%
6	7	98	6.3	100%	86.7%
7	8	124	8.3	100%	86.7%
8	9	153	10.8	100%	86.7%

In Table 4 we show the results for CDLL using the exhaustive white-box strategy. Our novel approach shows the same results of branch coverage and mutation score as the white-box approach.

5.1.1. Discussion

The results show that the black box approach is faster but lacks branch coverage and mutation score. On the other side, the white box approach using FCI and the exhaustive white-box achieve the same branch coverage and mutation score for the same

scopes. The second one takes considerably less time to generate the tests, but it produces a bigger suite.

5.2. Binary search tree

Figure 40

Binary search tree definition

```
class BST:
    root: "node"
    def __init__(self):
        self.root = None

class node:
    value: int
    right_child: "node"
    left_child: "node"
    parent: "node"
    def __init__(self, value: int = None):
        self.value = value
        self.left_child = None
        self.right_child = None
        self.parent = None
```

Figure 40 shows the definition of the classes that implement the data structure. We took the implementation from an open-source repository [19]. We generated test for the following methods:

- **insert(self, value: int):** Inserts a new node in the tree with the value "value".
- **delete_value(self, value: int):** Deletes a node in the tree with the value "value" if it exists.
- **find(self, value: int):** It searches for a node with the value "value" and returns it if it exists.
- **height(self):** Calculates and returns the height of the tree.

Table 5*Black-box statistics for binary search tree*

Node Limit	RepOK structures	Tests	Time (secs)	Branch Coverage	Mutation Score
1	2	14	0.8	67%	57.5%
2	4	40	3.2	88%	78.1%
3	9	120	11.2	99%	93.2%
4	23	386	41.2	100%	94.5%
5	65	1310	185.4	100%	94.5%
6	197	4610	535.9	100%	94.5%

In Table 5 we show the results for BST using the black-box strategy. It achieves the higher branch coverage (100%) and mutation score (94.5%) with a scope of four nodes.

Table 6*White-box statistics for Binary search tree*

Node Limit	Tests	Time (sec)	Branch Coverage	Mutation Score
1	14	2.5	67%	57.5%
2	35	20.1	88%	78.1%
3	88	148.6	99%	93.2%
4	184	2590.3	100%	94.5%

In Table 6 we show the results for BST using the white-box strategy with FCI. With a scope of 4 nodes, it achieves a 100% of branch coverage and a 94.5% of mutation score. However, as higher the scope, the execution time grows considerably faster. It cannot reach higher scopes because of timeout (> 1 hour).

Table 7*Exhaustive White-box statistics for Binary search tree*

Node Limit	RepOK structures	Tests	Time (sec)	Branch Coverage	Mutation Score
1	2	14	0.8	67%	57.5%
2	4	46	3.7	88%	78.1%
3	9	156	15.6	99%	93.2%
4	23	547	61.3	100%	94.5%
5	65	1951	240.1	100%	94.5%
6	197	7058	963.0	100%	94.5%

In Table 7 we show the results for BST using the exhaustive white-box strategy. It achieves a branch coverage of 100% and a mutation score of 94.5% with a scope of four nodes.

5.2.1. Discussion

In this study case, the three strategies achieve a branch coverage of 100% and a mutation score of 94.5%. The execution time of the white-box approach using FCI grows much faster than the others; it takes over 2500 seconds to generate the test suite for a four node scope. Nonetheless, the size of the test suite generated by the white-box technique is smaller and achieves the same coverage and mutation score than the others.

5.3. AVL

Figure 41

AVL definition

```

class AVLTree:
    root: "node"

    def __init__(self):
        self.root = None

class node:
    value: int
    height: int
    right_child: "node"
    left_child: "node"
    parent: "node"

    def __init__(self, value: int = None):
        self.value = value
        self.left_child = None
        self.right_child = None
        self.parent = None
        self.height = 1

```

Figure 41 shows the definition of the classes that implements the data structure. We took the implementation from an open-source repository [19]. Our tool has found a bug in the method that deletes a node from the tree. For this reason, we excluded it from the measurement. This bug causes some tests to fail and thus, *mutmut* could not calculate mutation score (as soon as a test fails, it stops calculating mutation score). Therefore, we generated test for the following methods:

- **insert(self, value: int):** Inserts a new node on the tree with the value "value".
- **find(self, value: int):** Searches for a node with the value "value" and returns it if exists.
- **height(self):** Calculates and returns the height of the tree.

Table 8*Black-box statistics for AVL*

Node Limit	RepOK structures	Tests	Time (secs)	Branch Coverage	Mutation Score
1	2	10	0.7	56%	37.0%
2	4	28	2.6	85%	73.2%
3	5	39	6.0	85%	73.2%
4	9	91	16.3	90%	80.6%
5	15	181	48.5	92%	88.9%
6	19	249	120.4	92%	88.9%

In Table 8, we show the results for AVL using the black-box strategy. It achieves a maximum branch coverage of 92% and mutation score of 88.9%. It reaches these bounds with a scope of five nodes.

Table 9*White-box statistics for AVL*

Node Limit	Tests	Time (sec)	Branch Coverage	Mutation Score
1	11	2.0	56%	37.0%
2	30	106.3	94%	85.2%

In Table 9, we show the results for AVL using the white-box strategy (with FCI). It only reaches a scope of two nodes because with higher scopes it exceeds the time limit (> one hour). With only two nodes as scope, it reaches a branch coverage of 94% and a mutation score of 85.2%.

Table 10*Exhaustive white-box statistics for AVL*

Node Limit	RepOK structures	Tests	Time (sec)	Branch Coverage	Mutation Score
1	2	10	0.7	56%	37.0%
2	4	32	3.5	94%	85.2%
3	5	47	7.6	94%	86.1%
4	9	123	23.0	97%	87.0%
5	15	259	58.8	99%	94.4%
6	19	367	187.3	99%	94.4%

In Table 10, we show the results for AVL using the exhaustive white-box strategy. The maximum branch coverage and mutation score are 99% and 94.4%, respectively.

5.3.1. Discussion

In this study case, the exhaustive white-box strategy achieves the higher branch coverage and mutation score. The black-box approach performs faster, but it cannot cover some parts of the code, and thus achieves a lower mutation score. The time to generate the test suite using the white-box approach with FCI is much higher than the others, reaching a scope of only two nodes. Higher scopes cause this last strategy to exceed the time limit, and thus it cannot cover parts of the code that require larger inputs.

5.3.2. A Bug Found

As we said before, PySEAT finds a bug in this AVL implementation [19]. The error occurs during some cases of node deletion. Apparently, after the deletion of a node

in the tree, the height of some nodes is not updated correctly, causing this field to be inconsistent with the rest of the structure. PySEAT finds the bug when exploring the method "delete_value" using a scope of four or higher. For this reason, the exhaustive white-box and black-box can find it, but the pure white-box strategy cannot (it only runs up to a scope of two nodes). Figure 42 shows one of the PySEAT's test cases that reveals the error.

The test fails on line 31, when it checks the class invariant after the execution of the "delete_value" method. Here, the class invariant does not hold because the height of the root is not properly actualized after the deletion. The regression assertion of line 35, automatically generated from PySEAT, shows that the current behavior is wrong; it expects the root's height value to be 3, while it should expect to be two.

Figure 42

Test that reveals a bug in an AVL implementation

```

def test_delete_value3():
    '''
    Self:
        0
       / \
      -1  1
     /
    -2

    Return:
        None
    End Self:
        0
       / \
      -1  1

    '''
    1: # Input Creation
    2: avltree0 = AVLTree()
    3: node1 = node(0)
    4: node1.value = 0
    5: node1.height = 3
    6: node1.parent = None
    7: node3 = node(0)
    8: node3.value = 1
    9: node3.height = 1
    10: node3.right_child = None
    11: node3.left_child = None
    12: node3.parent = node1
    13: node1.right_child = node3

    14: node2 = node(0)
    15: node2.value = -1
    16: node2.height = 2
    17: node2.right_child = None
    18: node4 = node(0)
    19: node4.value = -2
    20: node4.height = 1
    21: node4.right_child = None
    22: node4.left_child = None
    23: node4.parent = node2
    24: node2.left_child = node4
    25: node2.parent = node1
    26: node1.left_child = node2
    27: avltree0.root = node1
    28: # Method call
    29: returnv = avltree0.delete_value(-2)
    30: # Repok check
    31: assert avltree0.repok() → Fails here!
    32: # Regression assertions (captures the current behavior)
    33: assert returnv is None
    34: assert avltree0.root.value == 0
    35: assert avltree0.root.height == 3 → Wrong behavior!
    36: assert avltree0.root.parent is None
    37: assert avltree0.root.right_child.value == 1
    38: assert avltree0.root.right_child.height == 1
    39: assert avltree0.root.right_child.right_child is None
    40: assert avltree0.root.right_child.left_child is None
    41: assert avltree0.root.left_child.value == -1
    42: assert avltree0.root.left_child.height == 1
    43: assert avltree0.root.left_child.right_child is None
    44: assert avltree0.root.left_child.left_child is None

```

Note. PySEAT's test that reveals a bug on the `delete_value` method from the AVL implementation [19]

5.4. PEF vs PySEAT

First, it's important to have a clear understanding of the difference between both tools. PEF is a verification tool, it executes the target program using symbolic execution and checks if the behavior is the expected. PEF does not generate test cases and does not support complex data structures (structures containing aliasing constraints between objects, like a doubly linked list). On the other hand, PySEAT explores the target using generalized symbolic execution. It analyzes Python programs and checks whether the execution of methods satisfy the class invariant. It also generates the test cases for the

explored paths of the SUT. If it finds out that a method violates the class invariant, it will generate a test case that reproduces the error. For these reasons, we contrast the performance of both tools in two cases. On one side, we use a simple user-defined structure, a singly linked list. On the other side, we use a more complex data structure, a binary search tree with parent references.

5.4.1. Singly Linked List

Figure 43

Singly linked list definition

```

class LinkedList:
    head: "Node"

    def __init__(self):
        self.head = None

class Node:
    elem: int
    next: "Node"

    def __init__(self, elem: int):
        self.elem = elem
        self.next = None

```

Figure 43 shows the definition of the classes that implement the data structure. We compared both tools over the following methods:

- **is_sorted(self):** Returns true if the list is in strictly ascending order. False otherwise.
- **swap_node(self):** Swaps the first two nodes of the list if they are not in order.
- **delete(self, elem: int):** Deletes the first node on the list with the value "elem".
- **append(self, elem: int):** Inserts a node with the value "elem" at the end of the list.
- **prepend(self, elem: int):** Inserts a node with the value "elem" at the beginning of the list.

Table 11 presents the results we get from the execution of both tools over this case. PEF generates inputs up to 9 nodes, so to get equality of conditions, we use the parameter **max_nodes = 9** for the PySEAT execution. As PEF doesn't generate test cases, we can't measure mutation score. For the same reason, we measure branch coverage of PEF's execution, while on PySEAT we measure the coverage of the test suite it generates.

Table 11

Execution results of PEF and PySEAT over an acyclic singly linked list

Tool	Time (seconds)	Branch Coverage	Generates test cases?
PEF	28.3	100%	NO
PySEAT (max_nodes = 9)	6.7	100%	YES: 139 test cases

There is a considerable difference in the execution time; PySEAT takes only 6.7 seconds, and it also writes test cases to disk. The branch coverage is 100% for both tools.

5.4.2. Binary Search Tree

Table 12

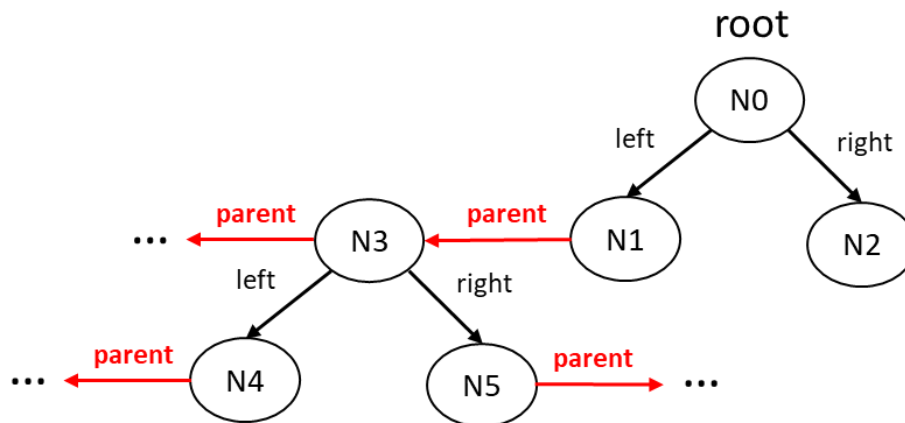
Execution results of PEF and PySEAT over a binary search tree

Tool	Time (seconds)	Branch Coverage	Generates test cases?
PEF	556	0%	NO
PySEAT (max_nodes = 4)	61.3	100%	YES: 547 test cases

We use the same implementation of section 4.2. The results of PySEAT are the same that table 7 shows. For a scope of only four nodes, it achieves a 100% of branch coverage and generates the corresponding test cases. On the other side, PEF couldn't explore any path of the target. PEF cannot build valid instances of a binary search tree with parent references. Thus, it cannot explore any path of the methods under test. The PEF algorithm to create heap allocated structures described in chapter one cannot handle complex data structures. For cases like this one, all the structures it creates are invalid because PEF cannot satisfy the constraint that a child in the tree must point to the parent in the "parent" field. For instance, figure 44 shows an example of an invalid instance that PEF creates for the binary search tree. As PEF always initializes fields to new nodes, the parent field of nodes never points to the actual parent.

Figure 44

Invalid BST instance created by PEF



Chapter 6

Related work

This work is based on PEF [3], an existing symbolic execution engine for Python programs. PEF allows the symbolic execution of Python independent functions and uses a contract system to specify programs preconditions and postconditions. PEF does not generate test cases. As shown in our experiments (Section 5.4.2), it does not work very well with heap-allocated data structures containing aliasing. Another Python symbolic execution tool is PyExZ3 [6]. The tool performs symbolic execution of Python programs that use only primitive data types. It neither generates test cases and, as PEF and PySEAT, it uses the Z3 constraint solver [12].

PEX [11] is an efficient automated test case generator for the .NET platform based also on symbolic execution using Z3 constraint solver. Nevertheless, PEX does not generate inputs automatically for non-primitive data types. For complex heap-allocated data structures, PEX requires the user to provide object factories: specific inputs that will be used during the symbolic execution of the code to generate tests.

Korat is a technique for automatic test input generation: given a predicate and a bound on the size of its inputs, Korat generates all the bounded inputs for which the predicate returns true. Korat exhaustively explores the bounded input space of the predicate but does so efficiently by monitoring the predicate's executions and pruning large portions of the search space [7]. Unlike korat, PySEAT's exhaustive white-box strategy is exhaustive only regarding the reference fields but not regarding primitive ones;

after generating all the bounded "shapes" of the structure, it uses symbolic execution to generate values for the primitive fields of the shape in order to cover branches in the SUT's code.

Chapter 7

Conclusion

In this work, we presented and developed a novel automated test generation tool for Python programs, called PySEAT. It generates test cases for implementations of complex objects (i.e. data structures containing aliasing) without user intervention; the process of the test suite generation is fully transparent to the user. The hearth of our tool is a novel engine implementing the generalized symbolic execution technique [1]. As far as we know, this is the first tool implementing this approach for the Python language.

We have implemented three input generation strategies: one based on white-box testing, another based on black-box testing, and a third variant based on the previous two strategies. The original white-box approach [2] requires that the user implements a particular type of class invariant to work, conservative class invariants. To the best of our knowledge, there is no sound algorithm to automate the creation of such class invariants from any given class invariant. Therefore, to automate this process, preventing the user of implementing a conservative class invariant and allowing the tool to work with any Python class invariant (also called repOk), we used what we call fully conservative class invariants (FCI) in the PySEAT's white-box implementation. The costs in terms of efficiency of using FCI were very high. The black-box input generation strategy [2] had better performance, but usually lacked branch coverage of the code. To solve the problems of both black-box approach and white-box approach with FCI, we implemented a third strategy, the exhaustive white-box.

We have assessed the three strategies with study cases taken from open source repositories. The results show that the black-box approach works faster than the others, but in several cases it lacks code coverage because it does not take into account the structure of the source code, decreasing the likelihood of finding errors. The white-box approach with FCI achieves high code coverage and mutation score with a smaller test suite than the others. The major problem it has is that it is inefficient. For complex data structures, its execution time may be prohibitive. It only performs well on very little inputs and thus, the likelihood of finding error decreases when more complex inputs are needed. The primary cause of its inefficiency is the use of fully conservative invariants to prune the spurious structures that lazy initialization generates. Frequently, the fully conservative invariant cannot filter all invalid instances, and this generates several problems: Infinite loops and stack overflows that the engine must handle, useless computations on spurious instances, and the need of a mechanism to remove invalid instances before writing test cases. Handling these problems, especially the last, is a very slow process. The exhaustive white-box approach solves all these problems by dealing directly with the root cause: The use of fully conservative invariants. Instead, the exhaustive white-box approach performs GSE of the class invariant to solve structural constraints before exploring the target method. It is similar to the black-box approach, but there is a key difference: It uses the partially symbolic instances that result from the exploration of the class invariant to perform traditional symbolic execution of the target method. The results show that the exhaustive white-box approach achieves the same branch coverage and mutation score than the white-box strategy with FCI, with considerably lower execution time but with a considerably bigger test suite. The size of the test suite that the exhaustive white-box technique generates is its major downside; it generates test suites that are too big, with multiple test cases exercising the same paths of

the code. On the other hand, the white-box strategy is ideal in this aspect, the test suites it creates are smaller and each test case covers exactly one path in the GSE tree of the program. As the major problem of PySEAT's white-box implementation is the use of FCI to validate lazy initialization steps, finding a more efficient mechanism of validating lazy initialization steps can cause the white-box technique to be a better alternative than the exhaustive white-box approach.

We also compared PySEAT's exhaustive white-box strategy with PEF (Section 5.4). PEF is only a verification tool for Python programs (it doesn't generate test cases). The results show that PEF cannot analyze programs using more complex data structures, like binary search trees with parent references. Because of the way it works, it is expected that PEF would not work on more complex structures like AVLs, Red-Black Trees, etc. Also, on acyclic singly linked lists (a simple heap-allocated structure that does not contain aliasing), PySEAT has shown better performance than PEF.

PySEAT uses the exhaustive white-box technique by default. We have found a bug with PySEAT using this strategy in the following open-source AVL implementation [19]. The error occurs during some cases of node deletion. After deletion it doesn't update correctly the height of some nodes, causing this field to be inconsistent with the rest of the structure. PySEAT finds the bug when exploring the method "delete_value" using a node's limit of four or higher.

7.1. Future work

There are many features that could be interesting to add to PySEAT in future works:

- **Support for more primitive types:** The tool currently supports only integers and booleans primitive types. Adding other types such as strings and arrays would allow PySEAT to support more Python programs.
- **Support for independent functions:** Currently, PySEAT only supports test generation for methods from a class.
- **Support method preconditions and postconditions:** It's possible to add a contract system to support method preconditions and postconditions. PySEAT could take advantage of the information of the precondition to avoid unnecessary explorations. The tool also can use the information of postconditions to create test oracles.
- **Usage of the methods from the API of a class to create the test input in test cases:** Currently, PySEAT test cases create the input using only constructors and field assignments. This causes the test case to be longer and harder to understand. A nice feature would be to PySEAT to write test cases that create the input using the methods from the API of the class. This would allow the test cases to be shorter and more readable.
- **Replace fully conservative invariants in white-box technique:** To avoid the user having the responsibility of creating a conservative invariant, PySEAT's white-box technique allows the use of any kind of class invariant and runs it in a fully conservative way to validate lazy initializations. This causes the procedure to be very inefficient. Finding a more efficient mechanism of validating lazy initialization steps can cause the white-box technique to be a better alternative than the exhaustive white-box approach because it generates smaller test suites that achieve the same code coverage and mutation score.

- **Concurrent executions:** PySEAT currently performs a sequential exploration of the target program. By exploring the different program paths concurrently, the execution time can be reduced drastically.

References

- [1] Khurshid S., Păsăreanu C.S., Visser W. (2003) Generalized Symbolic Execution for Model Checking and Testing. In: Garavel H., Hatcliff J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2003. Lecture Notes in Computer Science, vol 2619. Springer, Berlin, Heidelberg.
- [2] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test input generation with java PathFinder. SIGSOFT Softw. Eng. Notes 29, 4 (July 2004), 97–107.
- [3] Damián Barsotti, Andrés M. Bordese, and Tomás Hayes. (2018) PEF: Python Error Finder. Electronic Notes in Theoretical Computer Science, Volume 339, 2018, Pages 21-41.
- [4] Alessandro Bruni, Tim Disney, Cormac Flanagan. (2011). A Peer Architecture for Lightweight Symbolic Execution. University of California, Santa Cruz.
- [5] N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser and M. F. Frias, "BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support", in IEEE Transactions on Software Engineering, vol. 41, no. 7, pp. 639-660, 1 July 2015.
- [6] Thomas Ball and Jakub Daniel. (2015). Deconstructing Dynamic Symbolic Execution. Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence, Boston, MA, January 2015.

- [7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In Proc. International Symposium on Software Testing and Analysis (ISSTA), July 2002.
- [8] James C. King. (1976). Symbolic execution and program testing. *Communications of the ACM*, volume 19, número 7, pages 385–394.
- [9] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA '07). Association for Computing Machinery, New York, NY, USA, 815–816. DOI:<https://doi.org/10.1145/1297846.1297902>
- [10] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). Association for Computing Machinery, New York, NY, USA, 416–419. DOI:<https://doi.org/10.1145/2025113.2025179>
- [11] Tillmann, Nikolai & Halleux, Jonathan. (2008). Pex-white box test generation for .NET. 134-153. 10.1007/978-3-540-79124-9_10.
- [12] Leonardo Mendonça de Moura, Nikolaj Bjørner: Z3: An Efficient SMT Solver. TACAS 2008: 337-340.
- [13] Cristian Cadar, Daniel Dunbar, Dawson R. Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008: 209-224.

- [14] P. Ammann, J. Offutt. Introduction to Software Testing. 2nd. Edition, Cambridge University Press, 2017.
- [15] Model Checking Programs". W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. Automated Software Engineering Journal. Volume 10, Number 2, April 2003.
- [16] Godefroid, P., Klarlund, N., Sen, K. DART: directed automated random testing. Programming Language Design and Implementation, páginas 213–223. ACM (2005).
- [17] Python Documentation. <https://www.python.org/doc/>
- [18] Python properties. <https://docs.python.org/3/howto/descriptor.html#properties>
- [19] Repository of AVL and BST study cases.
https://github.com/bfaure/Python3_Data_Structures
- [20] Ariane's 5 explosion.
<http://www.users.math.umn.edu/~arnold/disasters/ariane5rep.html>
- [21] NASA's Mars climate orbiter.
https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf
- [22] Coverage. <https://pypi.org/project/coverage/>
- [23] Mutpy. <https://pypi.org/project/MutPy/>

Appendix A

Usage of PySEAT

In this section we explain how to use the tool, the parameters it takes, where it saves the output, and the existing tools that PySEAT uses to measure branch coverage and mutation score.

A.1. Installation

First, clone this repository to your local machine using git:

```
git clone https://github.com/JuanmaCopia/PySEAT
```

Second, create and activate a virtual environment on the project folder:

```
cd PySEAT/  
python3 -m venv env  
source env/bin/activate
```

Finally, Install the requirements:

```
sudo pip install -r requirements
```

After those commands, PySEAT should be ready to run. To verify everything works properly, we can run the test cases with:

```
python run_tests.py
```

A.2. Parameters

PySEAT takes as argument three obligatory parameters:

Table 12

PySEAT's required parameters

Name	Type	Function
filepath	String	Indicates the path to the python module that contains the SUT
class_name	String	Indicates the name of the class under test.
methods	String	Indicates the methods to test from the class.

We can also provide some optional parameters to the tool. When they are not provided, the default value is used. The following table describes all parameters, showing its type and default value:

Table 13

PySEAT's optional parameters

Name	Type	Default value	Function
max_nodes	Int	5	Limits the maximum amount of "nodes" or "new instances" that PySEAT can use to generate instances of the class under test.
max_depth	Int	10	Limits the maximum depth of the exploration tree regarding conditional branch points. (PySEAT prunes paths that overpass this value).
timeout	Int	5	If the exploration of a SUT's path takes longer than this value, PySEAT prunes it.
coverage	Bool	False	Measures branch coverage of the test suite it generates using the tool <i>coverage.py</i> .
mutation	Bool	False	Measures mutation score of the test suite it generates using <i>mutpy</i> .
quiet	Bool	False	Displays less output.
run_tests	Bool	True	Runs the test suite it generates using <i>pytest</i> .
test_comments	Bool	True	Creates a comment on each test case representing the input and the output. It uses the method <code>__repr__</code> of each class.

blackbox	Bool	False	When it is true, PySEAT generates test cases using the black-box approach. Otherwise, it use Exhaustive white-box as default.
----------	------	-------	---

A.3. Execution

The easiest way to run PySEAT is through a configuration file. We can add a new section to the default configuration file in *PySEAT/config.ini*, describing the parameters we described in the previous section. We can also provide a custom .ini file.

Figure 46

PySEAT's configuration file example

```

; PySEAT/config.ini

[DEFAULT]
max_nodes = 5
max_depth = 10
timeout = 5
coverage = false
mutation = false
quiet = false
run_tests = true
test_comments = true
blackbox = false

[LinkedList]
filepath = tests/linkedlist/ll.py
class_name = LinkedList
methods = is_ordered,swap_node
max_nodes = 4

[CircularDoublyLinkedList]
filepath = tests/circulardoublylinkedlist/cdll.py
class_name = CDLinkedList
methods = insert_after,insert_before,delete,append,prepend

```

Figure 46 shows an example of a configuration file. It has two sections: `LinkedList` and `CircularDoublyLinkedList`. Each section describes a class under test, and PySEAT will generate the test suite for each of those classes. The default section contains the

default values and other sections can override them. Note that the "LinkedList" section overrides the *max_nodes* parameter.

To run PySEAT using the default config.ini file use the -c option:

```
python <path-to PySEAT/__main__.py> -c
```

We can also execute PySEAT as a module:

```
python -m PySEAT -c
```

To provide a custom configuration file, just add the path after the -c option using any of the previous ways:

```
python <path-to PySEAT/__main__.py> -c <path-to .ini file>
```

We can also pass the parameters to PySEAT through the command line arguments:

```
python <path-to PySEAT/__main__.py> <path-module-to-test.py>  
<class-name> -m <methods-names> [OPTIONAL ARG. 1] [OPTIONAL ARG. 2] ...
```

A.3.1. Measurement of branch coverage and mutation score

PySEAT uses a tool called *coverage* [22] to measure branch coverage. To measure mutation score, it uses a tool called *mutpy* [23]. These measurements are disabled by default but we can enable them by setting the parameters "mutation" and "coverage" to true.