Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 339 (2018) 21–41

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

PEF: Python Error Finder

Damián Barsotti[1] Andrés M. Bordese[2] Tomás Hayes[3]

Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba
Argentina

Abstract

PeerCheck [1] is a lightweight library-based approach to implement symbolic execution over object-oriented languages without modifying the analyzed code. Unlike traditional symbolic execution engines, the symbolic semantics is built as an external library that runs together with the target program. They can accomplish this task thanks to special features of the python programming language which allows one to implement symbolic values with class objects. However, their approach is limited to symbolic execution on primitive types. In this work, we show an extension that enables one to perform symbolic execution with user-defined class objects. In addition, we build a new verification tool, called PEF, which implements the described technique, and our extension. This tool can be used to verify programs written in the Python programming language, and we developed in python on top of the Z3 SMT solver. We used the tool to verify its own code, showing its potential and usefulness to test programs in production. We also evaluate the tool for testing well known algorithms and data structures.

Keywords: Reliability, verification, unit testing, symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation, testing Python programs.

1 Introduction

Symbolic execution is a popular technique for reasoning about programs. It implements a symbolic semantics that allows one to execute programs on symbolic inputs representing multiple concrete inputs, thus simultaneously covering many concrete executions on single (symbolic) execution paths.

Based on this technique, a variety of tools have been successful in detecting program errors automatically. Traditionally, the symbolic semantics was implemented either by:

- writing a new interpreter for the programming language; or
- translating the code into another language with symbolic execution support; or

[1] Email: damian@famaf.unc.edu.ar
[2] Email: andresb9163@gmail.com
[3] Email: tomyhayes@gmail.com

https://doi.org/10.1016/j.entcs.2018.06.003

- instrumenting the target code. That is, writing an external program that injects instructions into strategic locations to achieve symbolic support while maintaining the original interpreter.

For example, Java PathFinder [13] is implented as an interpreter for the JVM bytecode; DART [8] is an interpreter for x86 binary programs; Bitblaze [12] translates binary code to an intermediate language, and CUTE [11] implements C code instrumentation.

Unfortunately, these approaches have some drawbacks. They require a large amount of code that have to be developed and maintained. Also, they present problems when handling generated or dynamically loaded code. Finally, it may be difficult to follow the evolution of the language, and to support nonstandard extensions.

To solve these problems, PeerCheck [1] proposes a novel alternative. It introduces a *peer architecture*, where the symbolic execution engine works alongside the program. Rather than defining a new interpreter or compiler, it simply implements the symbolic execution engine as a library in the target language. The peer architecture exploits the capabilities provided by dynamic and pure object oriented languages. The idea is based mainly on their ability to dynamically dispatch primitive operators such as arithmetic operators, array accesses, conditional branch tests, etc.

However, the peer architecture has been criticized [14] since it only allows one to check programs implemented with primitive types. This paper shows that it is feasible to extend PeerCheck to handle user-defined class objects. The extension opens a new path to build verification tools with little effort and only restricted by the ability of the automated provers. Nonetheless, the power of these provers, like SMT solvers [7], is constantly advancing.

We present the technique applied to programs written in Python. Python is a pure object oriented language used to implement high level functionality over a wide variety of systems. However, achieving a high degree of coverage testing on Python programs has been recognized as a necessity due to the lack of static type checking in this language. Moreover, our methodology can by applied to other object oriented program languages.

In order to realize our approach, we implement the *PEF (Python Error Finder)* verification tool. The tool works as a proof of concept of the improved peer architecture technique mentioned above. It was developed by integrating the Z3 [6] SMT solver with our Python (version 3) code. The tool also implements a contract system for expressing pre and postconditions, program exceptions, and type restrictions.

The rest of the paper is organized as follows: In Section 2, we give an overview of the symbolic execution technique. Section 3 describes the PEF architecture and its subsystems. Section 4 presents the implemented contract system. Section 6 shows how the tool was used to verify its own code. Section 7 shows some results by applying the tool on well known algorithms and data structures. In Section 8, we discuss related work. Finally, Section 9 presents the conclusions and discusses

future work.

2 Symbolic Execution

Symbolic execution is a technique originally proposed by James C. King [10] in order to obtain reliable software programs. In this technique the program execution is emulated defining an alternative "symbolic execution" semantics by replacing its concrete input values by arbitrary symbols. Thus, the analyzed programs transforms symbolic states during its execution, instead of the real data objects. Symbolic execution attempts to cover all possible execution paths by choosing different satisfiable path constraints until all execution traces are fully explored.

Consider a program consisting only of assignment statements and conditional statements (`IF-THEN-ELSE`). One way to visualize this technique is representing programs as execution trees: each node will represent a execution statement, and each edge a transition from one statement to the next, according to the program's execution path. When the node represents a conditional statement, it will have two output edges, one for the `THEN` choice, and another for the `ELSE` choice.

The technique builds this tree from the root (program start) by executing the program symbolically and attaching each node to a symbolic state. A symbolic states includes the symbolic values of its variables α and a path condition PC. At beginning, the root node has $PC = True$, and α is the map from the input program variables $v_1, \cdots, v_n$ to the symbols $s_1, \cdots, s_n$ representing the initial symbolic values. Given a symbolic state $\sigma = \langle \alpha, PC \rangle$ attached to a node, the technique generates its child nodes as follows:

(i) Over a node representing an assignment statement, it creates a new child node with a new symbolic state. This has the same PC, and α is obtained by the replacing the left-hand side variable in every assignment statement by its right-hand side expression. This expression has to be first converted into a symbolic one, replacing its variables by the symbolic expressions of the variables defined on the map.

(ii) Over a node representing an "`IF` e `THEN` $S1$ `ELSE` $S2$" statement there are two possible path conditions. Thus, PC is updated to $PC = PC \wedge \sigma(e)$, and a new path condition $PC' = PC' \wedge !\sigma(e)$ is created. For these formulas, $\sigma(e)$ gives the symbolic value of the expression according to α. PC and PC' define at most two child nodes according to their satisfiability, by assigning concrete values to the associated symbolic variables. In order to decide the satisfiabiliy of the path restrictions, an automated prover or a custom decision procedure is used. If neither PC nor PC' are satisfiable, the symbolic execution for this path ends.

(iii) If the symbolic execution reaches an exit instruction or error (for example, the program fails or violates an assertion) the corresponding instance of symbolic execution ends, and an assignment that meets the current symbolic path condition is generated, using an automated prover, as in the previous case.

In step ii, if both path conditions are satisfiables, the method first chooses the `THEN` branch. Thus, the tree is generated in DFS order. In this case, the method arrive at a *free branch*, otherwise we say that is a *forced branch*.

As an example, Figure 1 shows the tree generated from the Figure 2 code. The edges are labeled with the corresponding line code number.

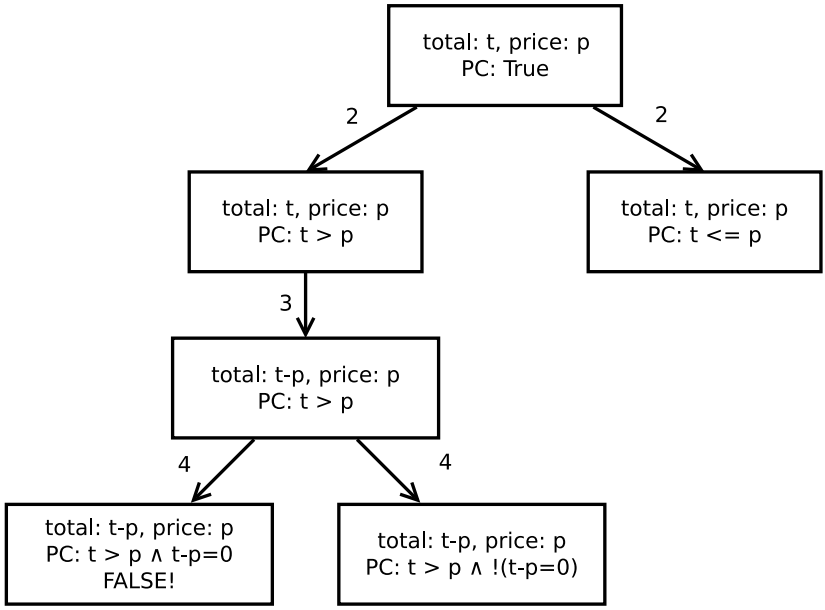

Fig. 1. Execution tree example.

```
1 def f(total, price):
2     if total > price:
3         total = total - price
4         if total == 0:
5             assert False
```

Fig. 2. Symbolic execution example.

If the program contains loops or recursive calls this technique can be generalized with the inconvenience that it can produce infinite trees. One solution, which ensures termination, is to limit the depth of the generated tree [10]. The solution is also useful when the size of the tree is large, to avoid scalability problems.

3 PEF Implementation

PEF follows the *peer architecture* proposed in [1]. The symbolic execution engine is implemented as a language library, which allows one its execution as a *peer* of the target program. This program is executed by the engine with special input arguments called *proxy objects* instead of concrete values. The proxy objects act as symbolic values for the symbolic execution technique.

When the target program performs a class operation on a proxy object, the interpreter dynamically dispatches a function call to the symbolic execution engine. This allows one to keep track of the input changes at run-time. In the particular case of a conditional jump that requires the value of a proxy object, the symbolic execution engine is reported again, and it queries an external prover (SMT solver Z3) to decide which route to explore. In Figure 3 we can see this architecture.

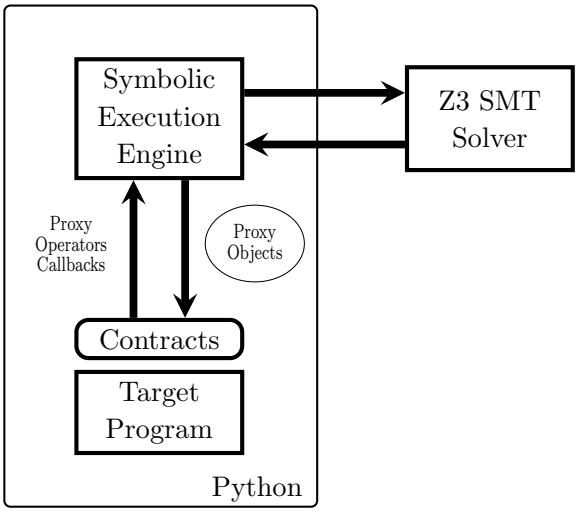

Fig. 3. PEF Architecture.

The symbolic execution engine is implemented by two coupled subsystems: the *execution loop* is in charge of running the target program several times through different tree execution paths, and the *proxy objects*, one for each type class, are responsibles for tracking changes of the symbolic values.

3.1 Execution Loop

The subsystem is implemented by the `explore` method. Figure 4 shows its simplified code. It takes the target program and the proxy objects (symbolic inputs) as arguments. Each iteration in the loop (line 8) explores a new path of the execution tree, and it stores the return value, or throw exception if any (line 12 to 14). It also captures the output of the program by intercepting calls to the `print` function.

Whenever the system reaches a free branch, it chooses one path to explore, and it schedules a future analysis for the other path. In order to register them, the subsystem creates two global lists (stacks):

- `_path` (initialized on line 6)
  It is a list of boolean values that point out the chosen path in a free branch node. The boolean `_path[i]` indicates the current branch choice (`THEN` or `ELSE`) for the ith free branch in a path. So, the symbolic execution engine uses this list to schedule each path of exploration. Also, the same list is used for the full analysis of the target program.

```
1  def explore(function, args, kwargs):
2    """
3    Explores all reachable paths from 'function'
4    with initial 'args' y 'kwargs'.
5    """
6    _path = []
7    have_paths_to_explore = True
8    while have_paths_to_explore:
9      _pathcondition = []
10     result, error = None, None
11     try:
12       (result, output) = function(*args, **kwargs)
13     except Exception as e:
14       error = e
15     finally:
16       yield (args, kwargs, result, error,
17              output, _pathcondition)
18     while len(_path) > 0 and not _path[-1]:
19       path.pop()
20     if not _path:
21       have_paths_to_explore = False
22     else:
23       _path[-1] = False
```

Fig. 4. Simplified code of the `explore` function.

- `_pathcondition` (initialized on line 9)
  It contains the current path condition (PC) being explored. It is initialized by this subsystem, and then it is filled by the proxy objects, which we will show later. `_pathcondition[i]` stores the restriction of the ith free branch in the current path. These formulas are conjoined, and the result will be used by the prover to decide the satisfiability of the path condition.

These variables are a means of communication between the subsystem and the proxy objects. This allows the subsystem to execute all DFS paths of the execution tree: PEF uses the information stored in `_path` to obtain the prefix of the new path to be explore. At beginning of each iteration (line 9), the list `_pathcondition` is emptied. Then, the postfix of false values in `_path` is removed (line 19), and the last true value is negated. This process allows one to explore the negative branches. When `_path` is empty (line 20) all the branches where explored, and the process ends. Moreover, branches are explored to some depth or until the prover cannot decide the satisfiability of the expressions involved. PEF informs the user about the branches that are ignored.

Each time the symbolic execution finds a function call, it will be executed symbolically since its proxy objects parameters (if any) will inform PEF of the symbolic state change. The return value (usually symbolic) is used to continue the original execution.

The `explore` method was programmed as a generator (line 16), where for each explored path it returns a tuple containing:

- the list of positional and variable length proxy objects arguments (`args` y `kwargs`)

of the target program;

- the symbolic result of the execution path (`result`);
- the thrown exception, if any (`error`);
- the output written to `stdout` (`output`)
- and, the final path condition (`_pathcondition`).

The symbolic execution process have to be modified to ensure termination (Section 2). This was done by bounding the scanning depth of the execution tree. The default limit was taken from [1], but can be easily changed if necessary. This value was in practice sufficient for our analyzed example programs. More information about these results will be presented in Section 7.

3.2 Proxy Objects

Object-oriented languages, such as Smalltalk, Ruby or Python, represent all data as objects. In addition, its operations are translated into method calls. The translation is also performed on operations of primitive types such as `+`, `>`, `==`, `and`, etc. Moreover, the type of the objects is defined only by the name of its attributes and methods, regardless of whether it is a primitive or user defined type. This property of languages is known as Duck Typing [4]. Our work is based on these features: PEF executes the target program replacing its arguments with proxy objects, which have the same methods and attributes names. Therefore, proxy objects simulate the behavior of real objects, while tracking the execution of the program.

In order to simulate binary operations on primitive types, proxy objects have to mimic their implementation. Python implements a "double dispatch" mechanism for binary operations. For example, the expression `x + y` is translated to the call `x.__add__(y)`, if the class of `x` knows the class of `y` (e.g. when they belong to the same class). If `x` belongs to a built-in class (e.g. `int`) and `y` belongs to a different one, Python performs the alternative translation `y.__radd__(x)` (addition to the right) by delegating the result to the `y` class method. In case the class of `y` cannot resolve the operation, this will result in a run-time error. The double dispatch mechanism is used by Python in other classes, for example `bool`: the expression `a < b` has an initial translation `a.__lt __(b)`. However, if `a` and `b` belong to different classes, the operation is translated to `b.__gt__(a)`.

As we have already mentioned, proxy objects are in fact symbolic values. Besides, they perform the symbolic execution process in PEF. They act like "spies", which behave like other objects, emulating their behavior, and tracking all their operations. The automated prover uses the information collected by our proxy objects to decide which paths can be taken in the target program execution.

PEF implements different classes of proxy objects in order to represent values of different types. As an example, in Figure 5 we show a simplified version of the integer proxy class, while our implementation supports the same set of numeric operators and predicates as the Python built-in integer class.

[4] https://en.wikipedia.org/wiki/Duck_typing.

```
1  class IntProxy(ProxyObject):
2      """
3      Symbolic integers
4      """
5
6      def __init__(self, value=None):
7          if value is None:
8              self.term = smt.fresh_var(int)
9          else:
10             self.term = smt.make_symbolic(value)
11
12     def __add__(self, other):
13         """
14         x.__add__(y) <==> x+y
15         """
16         return IntProxy(smt.op('+', self.term,
17                                other.term))
18     def __radd__(self, other):
19         """
20         x.__radd__(y) <==> y+x
21         """
22         return self.__add__(other)
23
24     def __eq__(self, other):
25         """
26         x.__eq__(y) <==> x==y
27         """
28         return BoolProxy(smt.pred('=', self.term,
29                                   other.term))
30     def __req__(self, other):
31         """
32         x.__req__(y) <==> y==x
33         """
34         return self.__eq__(other)
```

Fig. 5. `IntProxy` class (abbreviated code)

The code shows the implementation for the constructor (method `__init__`), the addition operators (methods `__add__`, `__radd__`), and equality tests (`__eq__`, `__req__`). It include the global object `smt` as an interface with the prover.

We first describe the constructor method. The call to `smt.fresh_var(int)` (line 8) generates a fresh integer variable for the prover, thus creating a symbolic value. If an initialization value is passed, the constructor calls the method `smt.make_symbolic(value)` (line 10) which creates an integer constant instead. Besides, if the initialization value is already symbolic, it is stored unmodified.

Then, the `__add__` (line) and `__eq__` methods return proxy objects representing integer o boolean expresions used in the path conditions. They call `smt.op` and `smt.pred` to construct them.

The implementation of **boolean proxy objects** is more complex because it decides which branches to follow in the execution tree of the program. A simplified version of the class is shown in Figure 6. The implementation is linked to the execution loop (Section 3.1) through the global `_path` and `_pathcondition` lists.

```
1  class BoolProxy(ProxyObject):
2    """
3    Symbolic booleans
4    """
5    def __init__(self, formula=None):
6      if formula is None:
7        self.formula = smt.fresh_var(bool)
8      else:
9        self.formula = smt.make_symbolic(formula)
10
11   def __not__(self):
12     return BoolProxy(smt.formula('!', self.formula))
13
14   def __bool__(self):
15     # Test path conditions
16     true_cond = smt.solve(smt.formula('&',
17                           _pathcondition, self.formula))
18     false_cond = smt.solve(smt.formula('&',
19                           _pathcondition,
20                           smt.formula('!', self.formula)))
21     # There is only one path
22     if true_cond and !false_cond:
23         return True
24     if !true_cond and false_cond:
25         return False
26
27     if len(_path) > len(_pathcondition):
28       # Have to recover previous state
29       branch = _path[len(self._pathcondition)]
30       if branch:
31           _pathcondition.append(self.formula)
32       else:
33           _pathcondition.append.(smt.formula('!',
34                                              self.formula))
35           return branch
36
37     # Reachs the max limit of the exploration
38     if len(_path) >= MAX_DEPTH:
39       # Paramos la exploración de esta rama.
40       raise MaxDepthError()
41
42     # Follows the True path
43     _path.append(True)
44     _pathcondition.append(self.formula)
45     return True
```

Fig. 6. `BoolProxy` class (abbreviated code).

Each time a conditional predicate contains proxy objects, the `_ bool__` class method is called by the Python interpreter to decide which path have to take. Therefore, the prover is consulted for the feasibility of the paths. It decides whether the formula stored in the `BoolProxy` object or its negation are satisfied along with the rest of the formulas in `_pathcondition` (line 16 to 20). If only one of these set of formulas is satisfied (i.e. the program execution arrives to a forced branch), the method returns the boolean value of the associated branch to the interpreter

(line 22 to 25). So, the execution of the target program continues without modifying the lists `_path` and `_pathcondition` (line 16 to 25). Otherwise, when both set of formulas are satisfiables (i.e. the program execution arrives to a free branch):

(i) If `_path` contains more values than the length of the path condition, the execution arrived at an already explored free branch. In this case, the method appends the formula or its negation to the path condition, according the current truth value of the branch stored in `__path`. Finally, it returns to the interpreter the boolean value associated with the branch to be executed in the current run (line 27 to 35).

(ii) If the depth bound is exceeded, then this path is terminated by throwing an exception (line 38 to 40).

(iii) Otherwise, the branch was not explored. Thus, the true branch have to be explored on this test run, so this boolean value is returned. In addition, the value is added to `_path`, and the current formula is stored in the path condition (line 43 to 45). The False branch is deferred to a subsequent run. Note that the `_path` list is only modified here and by the execution loop.

3.3 Container Proxy Objects

Besides integers and booleans, PEF can perform symbolic execution with *list*, *slice* and *string* symbolic values. These are implemented with the proxy classes `ListProxy`, `StringProxy` and `SliceProxy`.

The class `ListProxy` implements indexing operation `l[i]`, list update `l[i]=j`, length `len(l)`, list concatenation `l+m`, replication `l*i`, insertion `l.insert(i,j)`, append `l.append(i)`, lexicographic comparison `l<m`, etc. Lists `l`, `m` and indices `i`, `j` can by also proxy objects.

The lists were represented in z3 as arrays. In addition, we had to represent the limit of its size by adding a symbolic integer, since the Z3 arrays are unbounded. In order to communicate this constraint to Z3 we add a global list called `_facts` (in addition to `_path` and `_pathcondition`). This list defines conditions to be assumed by the prover. Figure 7 shows the constructor method of the class `ListProxy`. In line 10 the method appends the non-negative initial length constraint.

The represented length constraint can generate new branches in a program execution tree. Figure 8 shows this behavior.

When the function is called with a symbolic list `ls`, the symbolic execution generates three different path conditions: `[len(ls)==0]`; `[len(ls)>0, ls[0]<0]`, and `[len(ls)>0, !ls[0]<0]` (line 2). In the first path PEF throw the `IndexError` exception since it cannot evaluate `ls[0]` in the condition. In the second path, the symbolic execution performs the list update (line 3). The last path corresponds to the `else` branch.

This `else` branch continues with a loop over the symbolic list (line 5). The first iteration is performed without branching since the path condition includes the constraint `len(ls) > 0`. In the following iterations, the symbolic execution branches a path: one path returns `None`, and the other continues in the loop. In

```
class ListProxy(ProxyObject):
    """
    Symbolic lists
    """
    def __init__(self, initial=None):
        if not initial:
            self._array = smt.fresh_var(array_int)
            self._len = smt.fresh_var(int)
            # Always positive lengths
            _facts.append(self._len >= 0)
        else:
            self._array = smt.make_symbolic(initial)
            self._len = smt.make_symbolic(len(intitial))
```

Fig. 7. `ListProxy` constructor class (abbreviated code).

```
def f(ls):
    if ls[0] < 0:
        ls[0] = 0
    else:
        for j in ls:
            print(j)
    return None
```

Fig. 8. Induced branches by list length (example code).

this last case, PEF appends the condition `len(ls) > i`, with `i` the number of the iteration. This process continues until the maximum exploring depth is reached.

The list proxy objects implemented in PEF cannot store values of different types since Z3 arrays have this restriction. Furthermore, our current implementation only allows us to store symbolic or concrete integer values. Anyway, concrete lists can store concrete or symbolic values with different types, since in this case, it is not necessary to represent them in Z3.

In order to support all operations on lists, it was necessary to implement intermediary *slices*. Slices are widely used since they allow the extraction of list segments, and they are widely used by Python programmers. For example, given the slice `a:b`, `ls[a:b]` is the sub-list from the position `a` (starting at 0) to the position `b-1`.

The *strings* and lists have a similar implementation. Strings were represented in Z3 with an array and a symbolic integer since Z3 does not provide native support for this type. Besides, the `StringProxy` class only has a subset of the functionalities provided by the `ListProxy` class, since strings are not mutable, unlike lists.

4 Contract System

In order to improve error detection, we incorporate a contract system which can define types and properties to be validate or impose on the program execution. Its syntax is non-invasive, and follows the *docstring* convention [5] . We define four kind

[5] https://www.python.org/dev/peps/pep-0257.

of contracts: preconditions, postconditions, types and exceptions.

The **precondition contracts** are used to impose initial constraint to the argument values of a function. They start with the keyword `:assume:`, and continue with some boolean expressions separated by commas, which are interpreted as a conjunction of formulas. Figure 9 shows an example.

```
def positive_division(i, j):
    """
    :assume: i > 0, j > 0
    """
    i // j
```

Fig. 9. Precondition contract (example).

Within the contract, the user can refer to any of the input arguments or functions defined in the same context of the target program.

Postcondition contracts are used to define the properties that have to be satisfied after the target program is run. These contracts are used by PEF as an oracle test to detect program errors. They start with the keyword `:ensure:`, and continues with some boolean expressions separated by commas, just like precondition contracts. Figure 10 shows an example.

```
def f(a, b):
    """
    :ensure: returnv >= a + b
    """
    return abs(a) + abs(b)
```

Fig. 10. Postcondition contract (example).

Within this type of contract the keyword `returnv` indicates the return value of the target program. The other variables only refer to the initial value of the execution.

Python is a dynamically typed language, and lacks explicit type definitions. In order to define the types of the program inputs, we extend the contract system with **type contracts**. These kind of contract allows us to generate the appropriate proxy object according the declared type. An example of type contract is shown in Figure 11.

```
def polimorfic_product(a, b):
    """
    :types: a: int, b: [int, str]
    """
    return a * b
```

Fig. 11. Type contract (example).

These contracts start with the keyword `:types:`, and continue with the type declaration. The syntax allows us to declare a list of types for each argument. In

this case PEF performs a symbolic execution process for each one. When there are several arguments that define more than one type, a symbolic execution process is run for each combination of possible type arguments.

Although type contracts are necessary for the symbolic execution of a program, there are cases in which PEF is able to infer the types automatically:

- When a class method refer to the variable `self`, PEF infers that its type is the class.
- When a program declares an argument as `*args`, PEF infers that it has type `list`.
- When a type contract defines an argument of type `list`, PEF infers that is a list containing integer values, since PEF is able to handle only this type.

If it is not possible to deduce the type of an argument, then PEF will throw an own contract type exception.

In Python not all exceptions point to an error, since they are sometimes used intentionally in programs to communicate their internal state. Therefore, the system includes **exception contracts**. These allow us to differentiate those uses of the exceptions.

Exception contracts begin with the keyword `:raises:`, continue with the type of the Python exception, and end with a list of conditions over the variables. If an exception of this type occurs in the symbolic execution of a path, PEF checks the validity of the conditions assuming the path condition. If they are true, the path is considered as correct. In Figure 12 we show an example where a division by zero is not considered an error.

```
def integer_division(i, j):
    """
    :raises: ZeroDivisionError: j == 0
    """
    return i // j
```

Fig. 12. Exception contract (example).

5 User defined proxy classes

The *peer architecture* proposal had some criticism [14] since it only allows one symbolic execution over primitive types. However, we will show how to extend this technique to generate symbolic values of user-defined class objects.

PEF implements the generation of proxy objects of primitive types or user-defined classes with an unique function called `proxify`. This function takes a type identifier, and returns a set of proxy objects. For presentation purposes, Figure 13 shows a simplified algorithm code.

At first, in line 2 the function asks whether its argument is a built-in type. In this case, it returns the corresponding proxy object. Thus, it returns an `IntProxy` object for `int` type, a `BoolProxy` object for `bool` type, etc.

```
1  def proxify(type):
2    if isBuiltin(type):
3      return proxify_builtin(type)
4    else:
5      # Get all type arguments in contract
6      argTypes = TypeContract.parse(type.__init__)
7      # Get all combinations of type arguments
8      argTypeCombinations = combine(argTypes)
9      result = []
10     for argTypesComb in argTypeCombinations:
11       # Proxify current combination of arguments
12       argProxyObjs = []
13       for typeArg in argTypesComb:
14         argProxyObjs += proxify(typeArg)
15       # Get all combinations of proxy arguments
16       argProxyObjCombinations = combine(argProxyObjs)
17       for argProxyObjsComb in argProxyObjCombinations:
18         # Symbolic execution of constructor
19         result += symbolicExec(type.__init__ ,
20                                argProxyObjsComb)
21     return result
```

Fig. 13. `proxify` method (abbreviated code).

Otherwise, if the argument is an user-defined class, the algorithm search for a type contract defined in its constructor (`__init__`) method (line 6). The types of the constructor arguments have to be specified in order to instantiate the proxy objects.

In line 8 the function generates all possible type combinations of types in the constructor arguments. This is because type contracts can specify a list of types for each argument, as mentioned in Section 4.

From line 9, the `result` variable accumulates the possible instantiated proxy objects from the input class. For that matter, the loop (next line) operates over each possible list of types of constructor arguments (variable `argTypesComb`). For each type in this list, the function performs a recursive call returning a set of proxy objects for each argument in the input class. The variable `argProxyObjs` accumulates the received objects, which can also be some instances of primitive types or user-defined classes. Note that, these recursive calls end since user-defined classes are eventually implemented with built-in types.

The function then generates all possible combinations of the proxy objects returned by its recursive calls (line 16). Each combination will be a set of symbolic arguments for the constructor of the input class. These are stored in the variable `argProxyObjCombinations`.

Some class object constructors (implemented in the `__init__` method) can induce several execution paths according to their arguments. This possibility will make it necessary to build some new instances of constructor arguments. For example, in Figure 14, the constructor creates different attributes `value` according to the `n` argument. Given this constructor, the function `proxify` returns two proxy objects: `s0` and `s1`, with `s0.value == i0` and `s1.value == i0+5`, where `i0` is the

proxy object created from the `n` argument by a recursive call.

```
class GreaterFive(object):
    def __init__(self, n):
        """
        :type: n: int
        """
        if n > 5:
            self.value = n
        else:
            self.value = n + 5
```

Fig. 14. Constructor paths (example).

In order to generate these proxy objects, the function symbolically executes the constructor of the input class (line 19), obtaining a proxy object of this class for each execution path. The symbolic execution uses as arguments of the constructor each combination of the proxy objects stored in `argProxyObjCombinations`. The generated proxy objects are accumulated in the `result` variable, and returned as a result of the function.

It should be noted that this algorithm has a limitation: only variables declared inside the constructor are instantiated to proxy objects. Therefore, user-defined classes modifying class variables or global variables cannot be fully instantiated to proxy objects. This only limits the coverage of the symbolic execution algorithms, possibly ignoring some execution paths of the target program.

6 Verification of PEF with PEF

Throughout the development of PEF, we find some problems implementing the proxy objects methods. This was because a poor documentation of the built-in types or the complexity of the involved operation. Therefore, we decided to apply this tool on itself. The result is a real use-case applied on a software in production showing the potential of PEF.

The task was performed using the tool as a library (PEF can also be executed in command line interface). The program is available with the PEF code (`implementation_test.py` file). It performs the symbolic execution of the proxy object methods implemented by PEF. Then, the symbolic result of each path is compared with the result of built-in operations in Python. The task is fully automatically performed:

(i) It searches for all implementations of proxy objects in PEF code. For each of them, it identifies the emulated built-in class. We call `c` each proxy class, and `real(c)` the built-in class emulated by `c`.

(ii) Given a class `c`, PEF explores the executions paths of its methods. For each method, PEF returns a set of tuples by performing symbolic execution. These tuples contain some input proxy objects, a symbolic result, a path condition, and a thrown exceptions if any (Section 3.1). Each tuple corresponds to an execution path.

(iii) With a path condition and a symbolic result, PEF can build a test case for the `real(c)` built-in method: it obtains a variable assignment that satisfies the path condition (obtains a model) by using the Z3 prover. This variable assignment is used as concrete arguments in the built-in method. Also, an expected concrete return value is obtained with the same assignment applied to the symbolic result.

(iv) The program runs the built-in method using the input arguments obtained in the previous step. If the built-in method and the execution of the path return the same exception, the path is considered *correct*. If the built-in method returns the generated expected value, the path is also considered *correct*. Otherwise the path has an error.

(v) If any test case of a method detects an error, then the implementation is considered incorrect.

We found several errors in PEF by using this program. For example, we have to read the C implementation of the function `indices` (used in the built-in slide class) in order to build the `SlideProxy` class. This code is part of the standard Python interpreter called *CPython*[6]. The misinterpretation of the code caused several errors in our implementation. Thanks to the program presented here we were able to detect them.

7 Results

We performed the evaluation of the tool with some algorithms. We conduct the evaluation on an Intel-based notebook computer, equipped with i5-4258U processor at 2.40GHz, 8GB ram at 1600Hz, and OSX 10.10.

7.1 QuickSort

Figure 15 shows the QuickSort algorithm implemented with list comprehensions.

We include in the code an optional precondition contract defining the list length. Although this contract is not mandatory, it was added to analyze the algorithm performance over different list sizes. We begin analyzing the number of explored paths, the depth of the execution tree generated, and the percentage of explored branches. In TABLE 1 we show these results.

As we see, the number of explored paths is the factorial of the list length, as expected. The depth of the execution tree is the number of free branches. This number is lower since its growth is quadratic with respect to the size of the list. The last column shows that it is only necessary to execute QuickSort with lists containing three elements to achieve full coverage of the branches.

TABLE 2 shows the run-time analysis for the same QuickSort code. The values were measured taking the average run-time over five executions.

[6] https://github.com/python/cpython.

```
def quicksort(l):
    """
    Quicksort using list comprehensions
    :types: l: list
    :assume: len(l) == 6
    :ensures: returnv == sorted(returnv)
    """
    if not l:
        return []
    else:
        pivot = l[0]
        lesser = [x for x in l[1:] if x < pivot]
        greater = [x for x in l[1:] if x >= pivot]
        lesser_sorted = quicksort(lesser)
        greater_sorted = quicksort(greater)
        return lesser_sorted + [pivot] + greater_sorted
```

Fig. 15. QuickSort code.

List Length	Explored Paths	Tree Depth	% Explored Branches
0	1	0	16
1	1	0	33
2	2	1	66
3	6	3	100
4	24	6	100
5	120	10	100
6	720	15	100

Table 1
QuickSort exploration results.

The table shows the run-time of the PEF analysis, and the QuickSort algorithm running as stand-alone program (concrete execution). The input arguments of the stand-alone executions are the lists returned by PEF as test cases: for each list size, PEF returns concrete input lists that cover the execution tree. We calculate the time of running the algorithm with all this test cases. This process is repeated five times and the results are averaged. The run-time analysis with PEF is also averaged over five times.

The table shows how both run-times increase rapidly with the list size. This behavior is due the fact that the number explored paths grows factorially with this size.

List Length	PEF Run-time [ms]	Stand-alone Run-time [μs]
0	165,29	195,59
1	309,68	185,1
2	641,28	196,24
3	2263,67	221,02
4	10610,27	398,12
5	62431,88	1522,04
6	$4{,}7 \cdot 10^5$	9430,11

Table 2
QuickSort run-time.

7.2 Other Results

We tested PEF over several programs, which are available with the PEF code (`examples.py` file). For each of these examples we tested correctness properties of the programs using the contract system. TABLE 3 shows the average run-time in seconds, over five executions.

The table shows the run-time of the PEF analysis, and the stand-alone programs (concrete execution). The input arguments of the stand-alone programs are the values obtained by PEF as test cases. These were calculated in the same way as the QuickSort algorithm (Section 7.1).

The execution times of PEF analysis are acceptable considering that tests where conducted on a standard notebook. Although, they are orders of magnitude bigger than those of the stand-alone programs. This overhead is a consequence of the intensive use of Z3 performed by PEF.

8 Related Work

The most successful symbolic execution tools such as KLEE [3], SAGE [9], Bitblaze [12], and S2E [5] are intended for static programming languages. However, dynamic languages such as Python, Perl or Java Script, are becoming popular since they allow programmers the creation of agile prototypes, and are easy to use.

The only mature tool, oriented to Python language, known for the authors is Chef [2]. This tool performs symbolic execution on the target program and the interpreter at the same time. Therefore, the tool needs a great processing power, not available in personal computers.

Another tool for performing symbolic execution in Python programs is Conpy [4]. This tool implements a dynamic symbolic execution method or *concolic testing* [11], but we do not find any reference to a functional software.

Program	PEF Run-time [s]	Stand-alone Run-time [s]
quicksort	470,01	9,43
fun	$3{,}13 \cdot 10^{-2}$	$3{,}7 \cdot 10^{-5}$
funab	$4{,}06 \cdot 10^{-2}$	$6 \cdot 10^{-6}$
funfor	1,31	$4{,}5 \cdot 10^{-5}$
funlist_in	0,37	$7{,}2 \cdot 10^{-5}$
funlist_iter	1,45	$3{,}08 \cdot 10^{-4}$
funlist_slice	31,71	$2{,}52 \cdot 10^{-4}$
assert_not_divisible	2,46	$1{,}57 \cdot 10^{-3}$
divisible	2,26	$1{,}33 \cdot 10^{-3}$
test_Test	0,21	$2{,}2 \cdot 10^{-5}$
test_mock_Test	0,2	$1{,}4 \cdot 10^{-5}$
test_gt	$4{,}69 \cdot 10^{-2}$	$4 \cdot 10^{-6}$
test_kw_args	$4{,}93 \cdot 10^{-2}$	$3 \cdot 10^{-6}$
multitypes	$2{,}48 \cdot 10^{-2}$	$7 \cdot 10^{-5}$
test_strings1	4,52	$6{,}8 \cdot 10^{-5}$
print_species	1,54	$5{,}82 \cdot 10^{-4}$
sum_lista	1,84	$1 \cdot 10^{-4}$
insertLeft	76,3	$1{,}65 \cdot 10^{-4}$
sum_fac	0,95	$6{,}3 \cdot 10^{-5}$
test_sum_fac	3,98	$1{,}33 \cdot 10^{-4}$
fact_spec	0,81	$1{,}4 \cdot 10^{-5}$
faulty_fact	0,82	$2{,}5 \cdot 10^{-5}$
suma_con_bug	0,11	$4{,}2 \cdot 10^{-5}$
f_l_j	0,42	$9 \cdot 10^{-6}$
power	0,79	$1{,}8 \cdot 10^{-5}$
abs1	$3{,}55 \cdot 10^{-2}$	$3 \cdot 10^{-6}$

Table 3
Average run-time of test examples.

9 Conclusion and Future Work

Peer architecture [1] is a promising technique for analyzing programs through symbolic execution. The proposal is based on the fact that pure object oriented languages can perform symbolic execution using its own defined objects as symbolic values. Therefore, the technique is implemented without modifying the program code neither the underlying interpreter or compiler.

Our work improves the technique by allowing the generation of symbolic values for user-defined class objects. This advance overcomes the problem described in [14]. The paper invalidates the technique since it only supports symbolic execution over primitive types. Therefore, the method was considered useless for real cases.

We also implement the technique building a tool called PEF (Python Error Finder). Thus, we apply the concepts of our work in order to automatically explore paths, and detect errors in Python programs. Besides, we implement a contract system which allows us to validate conditions and write restrictions over programs. These are written as noninvasive comments using Python conventions. The result is a proof of concept for the peer architecture including our extension. We show that this tool is powerful enough to verify its own code. PEF is available at `https://git.cs.famaf.unc.edu.ar/dbarsotti/pef`.

As future improvement of the technique we are interested in the implementation of symbolic global variables. Currently our tool only allows us to work with concrete values for these variables. Hence, there may be unexplored paths in the symbolic execution process.

In addition, the tool may be extended with variable aliasing support. However, there are often little practical consequences in using aliasing inside Python programs. In practice, it is rare for Python programmers to create an alias instead of a new variable since any assignment of a value to an identifier breaks the alias link.

Other improvements are conditioned by the power of the solver, such as the availability of decision procedures for new types or the ability to solve more complex expressions. However, we believe that the power of this tools will increase over time. This progress will allows us to analyze programs better, and in more depth.

Acknowledgment

We would like to thank Pablo Duboue and Clara Klimovsky for their generous help in correcting this work.

References

[1] Bruni, A., T. Disney and C. Flanagan, *A peer architecture for lightweight symbolic execution*, University of California, Santa Cruz, 2011.
URL `https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf`

[2] Bucur, S., J. Kinder and G. Candea, *Prototyping symbolic execution engines for interpreted languages*, in: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14 (2014), pp. 239–254.
URL `http://doi.acm.org/10.1145/2541940.2541977`

[3] Cadar, C., D. Dunbar and D. Engler, *Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs*, in: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08 (2008), pp. 209–224.
URL http://dl.acm.org/citation.cfm?id=1855741.1855756

[4] Chen, T., X.-s. Zhang, R.-d. Chen, B. Yang and Y. Bai, "Conpy: Concolic Execution Engine for Python Applications," Springer International Publishing, Cham, 2014 pp. 150–163.
URL http://dx.doi.org/10.1007/978-3-319-11194-0_12

[5] Chipounov, V., V. Kuznetsov and G. Candea, *S2e: A platform for in-vivo multi-path analysis of software systems*, SIGPLAN Not. **47** (2011), pp. 265–278.
URL http://doi.acm.org/10.1145/2248487.1950396

[6] De Moura, L. and N. Bjørner, *Z3: An efficient smt solver*, in: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08 (2008), pp. 337–340.
URL http://dl.acm.org/citation.cfm?id=1792734.1792766

[7] De Moura, L. and N. Bjørner, *Satisfiability modulo theories: Introduction and applications*, Commun. ACM **54** (2011), pp. 69–77.
URL http://doi.acm.org/10.1145/1995376.1995394

[8] Godefroid, P., N. Klarlund and K. Sen, *Dart: Directed automated random testing*, SIGPLAN Not. **40** (2005), pp. 213–223.
URL http://doi.acm.org/10.1145/1064978.1065036

[9] Kieyzun, A., P. J. Guo, K. Jayaraman and M. D. Ernst, *Automatic creation of sql injection and cross-site scripting attacks*, in: *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09 (2009), pp. 199–209.
URL http://dx.doi.org/10.1109/ICSE.2009.5070521

[10] King, J. C., *Symbolic execution and program testing*, Commun. ACM **19** (1976), pp. 385–394.
URL http://doi.acm.org/10.1145/360248.360252

[11] Sen, K., D. Marinov and G. Agha, *Cute: A concolic unit testing engine for c*, in: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13 (2005), pp. 263–272.
URL http://doi.acm.org/10.1145/1081706.1081750

[12] Song, D., D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," Springer Berlin Heidelberg, Berlin, Heidelberg, 2008 pp. 1–25.
URL http://dx.doi.org/10.1007/978-3-540-89862-7_1

[13] Visser, W., K. Havelund, G. Brat, S. Park and F. Lerda, *Model checking programs*, Automated Software Engineering **10** (2003), pp. 203–232.
URL http://dx.doi.org/10.1023/A:1022920129859

[14] Wang, R., P. Ning, T. Xie and Q. Chen, *Metasymploit: Day-one defense against script-based attacks with security-enhanced symbolic analysis*, in: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (2013), pp. 65–80.
URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/wang