A Peer Architecture for Lightweight Symbolic Execution

Alessandro Bruni Tim Disney Cormac Flanagan
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064
{abruni,tdisney,cormac}@ucsc.edu

Abstract

We present a novel and lightweight library-based approach to symbolic execution based on a *peer architecture*. Rather than defining a new interpreter or compiler to build a symbolic execution engine for a particular language, we simply use the existing features of the language so that the engine works as a *peer* of the target program. Our approach is based on the insight that languages that provide the ability to dynamically dispatch primitive operations (*e.g.* many scripting languages such as Python) allow us to track symbolic values at runtime. We present an architecture and implementation of our *peer architecture* in Python and discuss results of running our symbolic execution engine on several well known algorithms and data structures.

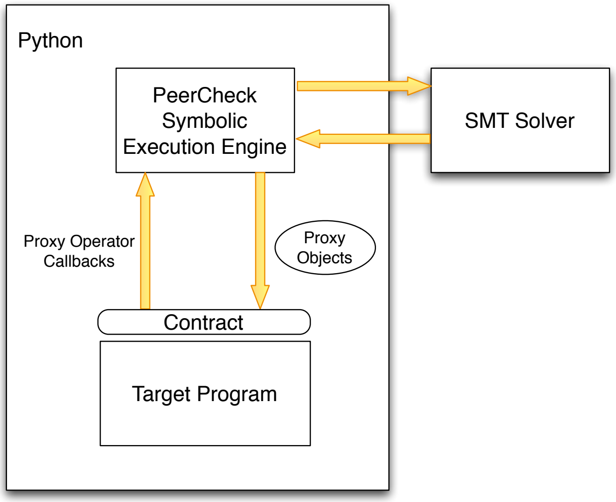
1. Introduction

Symbolic execution has emerged as a promising technique for reasoning about program behavior on multiple inputs simultaneously, and for exploring most or all possible program execution paths. A variety of recent tools have successfully used symbolic execution to detect defects and for automatic test input generation [1–3, 6, 7, 9, 10, 12, 15–17, 21, 22].

Traditional *concrete* execution, as implemented by typical language interpreters or compilers, executes the target program over a single input value such as a specific integer. In contrast, *symbolic* execution reasons about program execution using sets of integers (or sets of other values), where these sets are typically represented as constraints. Consequently, symbolic execution essentially involves defining a new *symbolic semantics* for the target language.

Previously, this non-standard symbolic semantics was implemented either by

Figure 1: Peer Architecture Diagram



- writing a new interpreter for the target language, or
- writing a new compiler/translator for the target language.

For example, Java Pathfinder [1] is an interpreter for JVM bytecode programs, and DART [6] is an interpreter for x86 binary programs.

Unfortunately, both of these approaches are somewhat heavyweight: they require significant amounts of code that must be developed and maintained; it may be difficult to track evolving language standards and to support non-standard language extensions; and there may be problems handling dynamically loaded and dynamically generated code.

This paper proposes an alternative *peer architecture* for symbolic execution, where the symbolic semantics is implemented as a *library in the target language*, rather than as a new implementation of the language. This peer architecture exploits extensibility capabilities provided by pure object-oriented languages, namely the ability to dynamically dispatch primitive operators such as arithmetic operators, array accesses, conditional branch tests, etc.

The peer architecture is shown in Figure 1. The symbolic execution engine is a library that runs within the same process as the target program. Instead of running the target program on concrete test inputs, the symbolic execution engine sends it special symbolic or *proxy* inputs instead. These proxy inputs are special kinds of values that allow the symbolic engine to observe how the target program manipulates these inputs.

When the target program performs an operation (*e.g.* "+") on a proxy, the language implementation dynamically dispatches to a callback provided by the symbolic engine, allowing the engine to track how proxy inputs influence other values in the program. When the target program performs a conditional branch that depends on a proxy input value, the symbolic engine is again called back, and it consults an external SMT solver to decide which branch to explore, and to ensure that the path explored by the target program always remains feasible. To obtain good coverage, the symbolic engine re-executes the target program multiple times, exploring a different execution path each time.

This peer architecture enables a rather substantial reduction in complexity over prior approaches. For example, Figures 5 and 6 below present an idealized symbolic interpreter that is only 50 lines of code. (Adding additional symbolic or proxy data types and associated operations would increase this size somewhat, but would still remain quite modest.)

Note that the benefits of this peer architecture are primarily in terms of simplicity rather than capability. That is, the architecture is not intended to facilitate additional kinds of symbolic execution, but rather to make it easier to implement, evolve, and deploy software tools based on symbolic execution. Alternatively, this architecture allows a more sophisticated symbolic engine to be implemented within a fixed complexity budget.

Our approach targets pure object-oriented languages such as Python, which is widely used to implement higher-level functionality in computer systems and web servers. Python facilitates an agile methodology with rapid prototyping of new functionality. Since Python lacks a static type system, good test coverage is critical, and so advanced testing tools are particularly beneficial in this domain. We note that security vulnerabilities are increasingly problematic in the 'scripting' layers of web servers, and cross-site scripting attacks have supplanted buffer overruns as the most common kind of security attack.

We also present a contract mechanism for expressing pre and postconditions on target routines, which help identify failing test runs. A failing precondition on the target routine identifies inappropriate test inputs and are uninteresting, whereas failing postconditions likely identify bugs, either in the target routine or in its specification (for example, an overly weak precondition or an overly strong postcondition). Our contract language also supports object invariants.

Figure 2: Absolute Value and Successor Functions

```
def abs(x):
    if x >= 0:
        return x
    else:
        return -x

def succ(x):
    if x == 42768:
        fail()
    return x + 1
```

Contributions: The central contributions of this paper are:

- It describes a novel *peer architecture* for lightweight symbolic execution (Section 4).
- It presents PEERCHECK, a symbolic model checker that provides a proof-of-concept for this architecture (Section 5).
- It presents a contract system for Python, where contracts serve as a test oracle for the model checker (Section 6).
- It shows experimental results from running PEERCHECK on several well known algorithms, demonstrating the viability of this approach (Section 7).

2. Review of Symbolic Execution

We begin with a brief review of symbolic execution, for readers not familiar with this idea.

Symbolic execution is a technique that uses symbolic values—instead of concrete values—to execute a software module. Symbolic values represent sets of possible real values; when a module is evaluated with symbolic values it produces symbolic expressions, which represent the range of possible values a concrete execution could produce.

To explore different paths through an algorithm with symbolic execution we add at each branch point a *path constraint* which is a symbolic expression that represents the conditions that must hold to follow that particular path. We can follow a particular path if the set of all path constraints (called the path condition) are satisfiable. Symbolic execution attempts to cover all possible execution paths by iteratively choosing different satisfiable path constraints until the space of reachable execution traces is fully explored.

Consider the implementation of the absolute value function shown in Figure 2. This function has two possible paths, which can each be followed by calling the function with a non-negative and a negative number. Instead of supplying concrete numbers to this function, we can interpret it symbolically by first following the true branch by constraining x to be non-negative (path condition $x \geq 0$) and then following the false branch by constraining x to be negative (path condition $x < 0$).

Other techniques exist to exercise many code paths, such as writing manual unit tests and testing random input values. However, manual testing requires a large amount of human effort to obtain good coverage, while random testing has difficulty finding bugs where only a few inputs causes failure, as is the case with the faulty successor function shown in Figure 2. Symbolic execution in this example would simply choose to add the constraint $x = 42768$ to follow the failing path, while random testing would have a 1 in 2^{32} chance of finding the bug (assuming 32 bit integers and a uniform distribution of random test inputs).

3. Operator Dispatch in Python

The primary language feature that enables our peer architecture is the ability to dynamically dispatch primitive operations. This dynamic dispatch allows us to create symbolic objects that mimic other values including "primitive" values such as numbers and booleans and record how they are used. This feature (or a similar one) can be found in a number of pure object-oriented programming languages, such as Smalltalk, Ruby, and Python. Since our implementation is written in Python, we discuss here how Python implements this feature.

Python represents all data as objects and all primitive operations (*e.g.* "+" or "-") are translated into method calls. For example, the expression `x + y` is replaced with `x.__add__(y)`. Alternatively, if `x` is a primitive integer but `y` is not, then this call in turn delegates to `y.__radd__(x)`. Since these methods can be defined and overridden, we can emulate "native values" such as integers and booleans by creating objects which implement the appropriate methods. Furthermore, this "double dispatch" mechanism allows proxy objects to be notified when they are either the left or right operand of a binary operator such as "+".

All other Python operators are dynamically dispatched in a similar manner. Comparison operations are also provided. For example, the equality test == is converted into a call to `__eq__` or its counterpart `__req__` (and again there are similar mappings to other comparison operations such as less than, greater than, etc.). By implementing this method our proxies can track when comparisons have occurred.

In Python all objects have a truth value (*e.g.* the number 0 is considered false and all other numbers are considered true). Whenever the truth value for an object is needed, the `__bool__` method is called. This feature allows us to also create proxies that emulate booleans.

To illustrate this dynamic dispatch mechanism, Figure 3 contains a simple implementation of complex numbers. It defines a `Complex` class that implements the `__add__` and `__radd__` methods, which allows complex numbers to be added using conventional operator syntax.

For example, consider the addition of two complex numbers, x and y. If we execute `x + y`, Python will interpret the statement as `x.__add__(y)`; calling this code will result in a new `Complex` object containing the result of the sum.

For the expression `x + 1`, the `__add__` method checks the type of the argument `other` (the integer `1` in this case) and, since it is not a complex number, adds the argument to the real component of x.

Figure 3: Complex Number Implementation

```
class Complex:
  def __init__(self, real, img):
    self.real = real
    self.img = img

  def __add__(self, other):
    if isinstance(other, Complex):
      result = Complex(self.real + other.real,
                       self.img + other.img)
    else:
      result = Complex(self.real + other,
                       self.img)
      return result

  def __radd__(self, other):
    return self.__add__(other)
```

Alternatively, the evaluation of `1 + x` results in a call of `1.__add__(x)`. Since the builtin integer is unaware of `Complex`, the addition is delegated to `x.__radd__(1)`. Our implementation of `__radd__` takes advantage of the commutative property of the addition to delegate the work to `Complex.__add__`.

This approach is a well known pattern in Python and other languages (like C++ with operator overloading) and is useful to extend the language's capabilities with other data types. In this paper, we apply this idea to extend the Python interpreter with symbolic execution capabilities.

4. The Peer Architecture for Symbolic Execution

At a high level, PEERCHECK works by calling a target program with proxy objects that emulate normal values. The proxies record how they are used during the program's execution. Once the target program returns, PEERCHECK uses the information recorded during that execution to produce a new path that can exercise new code branches next time the target program is called. See Figure 1 for a diagram of this architecture.

To begin symbolically executing the target program, PEERCHECK first creates a new proxy object for each parameter in the program. The proxies intercept and record all operations that are invoked on them and are transparent to the calling code.

There are two types of proxy objects, terms and formulas. Terms are used to represent symbolic values and stand-in for real values. Formula proxies are created when conditional statements (*e.g.* ==, <, <=, etc.) are evaluated with term proxies, and stand-in for boolean values.

After creating the term proxy objects, PEERCHECK invokes the target program. Whenever a conditional statement

Figure 4: SMT Solver Interface

$smt\_mkvar:$	$Unit \rightarrow Term$
$smt\_mkint:$	$Int \rightarrow Term$
$smt\_op:$	$String \rightarrow Term^* \rightarrow Term$
$smt\_pred:$	$String \rightarrow Term^* \rightarrow Formula$
$smt\_fop:$	$String \rightarrow Formula^* \rightarrow Formula$
$smt\_solve:$	$Formula \rightarrow Bool$

that involves a proxy is executed, a formula proxy is created. The formula proxy first checks the current path constraints to determine which branch it can follow by issuing a callback to the SMT solver. It then adds the chosen path constraint to the path condition.

For example, if the conditional statement involving the term proxy `x` was

```
if(x > 5)...
```

then the resulting formula proxy would check the satisfiability of `x > 5` against the current path constraints. Assuming the constraints were satisfiable then the proxy would add `x > 5` to the path condition and return `True`, causing the code to execute the then branch of the `if` statement.

Once the target program finishes, it returns control back to PEERCHECK. PEERCHECK then looks at the paths condition as recorded by the formula proxies and removes the last choices along with the respective constraints that need to be changed during the next execution of the target program.

The algorithm uses a depth limit to reduce the search space, following the assumption also made by the Haskell automatic testing library SmallCheck [14] that most bugs are reproducible by a small number of input values. By analogy, bugs can be found in a limited number of branch choices. To keep this bound low while reaching the necessary depth of choices involving symbolic variables, we distinguish between free branches and forced branches.

Free branches are branch points where an actual choice is made; free branches happen when both the true and the false branch are selectable. The default policy for free branches is to first explore the true branch and then, when the branch is fully explored at the desired depth limit, select and explore the false branch. *Forced branches* in contrast do not represent real choices, since the choice here is a forced consequence of earlier branch conditions. To increase the search depth we exclude forced branches from our depth count.

5. PEERCHECK Implementation

The PEERCHECK implementation is a simple python library built using the Z3 SMT solver [5] and taking advantage of Python's ability to do primitive operator dispatching.

Our implementation does not need to modify the python interpreter (unlike other symbolic execution engines), leading to a cleaner design. In particular:

1. No dedicated interpreter is needed, which avoids developing the set of components required to build an interpreter, such as parsers, etc.
2. The semantics are preserved. Languages such as Python do not have a formally specified semantics and the implementations are subject to change between releases. Since we do not have a separate interpreter we avoid much of the risk of discrepancies with the original language.

In this section, we present an idealized version of our PEERCHECK implementation.

5.1 SMT Interface

We begin by presenting a simplified interface to an SMT solver: see Figure 4. The interface provides the following functions.

`smt_mkvar` creates a new symbolic variable in the SMT solver

`smt_mkint` creates a new constant integer that can be used in an expression

`smt_op` applies an operation $(+, -, *, /)$ to one or more terms (which can be symbolic variables or constants or composite terms) and produces another term which represents the operation on the subterms

`smt_pred` applies a predicate $(=, <, \leq, >, \geq)$ to one or more terms and produces the respective formula

`smt_fop` applies a formula operation $(\wedge, \vee, \neg)$ to one or more formulas and produces the composite formula

`smt_solve` takes a formula and returns true if it is satisfiable, false otherwise

5.2 Integer Proxies

With this interface and using the proxy pattern described in Section 3, we can encapsulate symbolic values and have them behave like other Python data types such as integers. In particular, Figure 5 shows an implementation of a symbolic integer proxy with support for the addition operator (via the methods `__add__` and `__radd__`) and equality tests (via the methods `__eq__` and `__req__`). Note that, for presentation purposes, this class is abbreviated, but our actual implementation supports the whole set of numeric operators and predicates. The constructor takes in an SMT term to initialize the term field of the object, thus creating a symbolic variable for the interpreter.

Consider the expression `x + 1 == y`, where `x` is an `IntProxy` representing an underlying SMT term T and `y` is the builtin integer variable 42. The evaluation of this expression starts by calling `x.__add__(1)`, which creates the term $T + 1$ via appropriate calls to the SMT interface, and them returns a new `IntProxy` (called `z`, say) representing this term.

Figure 5: Symbolic Execution Engine: Integer Proxies

```
def __add__(self, other):
  # Return a new IntProxy representing the sum of the subterms
  #    self.term + other.term
  # Wraps 'other' in an IntProxy if it's not a symbolic value
  if type(other) == int: other = IntProxy( smt_mkint(other) )
  return IntProxy( smt_op('+', self.term, other.term) )

def __radd__(self, other):
  # Reverse side operator, calls __add__ taking advantage of the
  # commutative property for the sum
  return self.__add__(other)

def __eq__(self, other):
  # Returns a BoolProxy that wraps the equality formula:
  #    self.term == other.term
  # Wraps 'other' in an IntProxy if it's not a symbolic value
  if type(other) == int: other = IntProxy( smt_mkint(other) )
  return BoolProxy( smt_pred('=', self.term, other.term) )

def __req__(self, other):
  # Reverse side operator, takes advantage of the commutative
  # property and calls __eq__
  return self.__eq__(other)
```

Then `z.__eq__(y)` is called. This method call creates a symbolic SMT formula representing $T + 1 = 42$, and then returns a new `BoolProxy` representing this formula.

Thus, `IntProxy`s simply build up SMT terms and formulas that represent how they are used in the program. `BoolProxy`s are similar but somewhat more complex, since they need to react appropriately to calls to their `__bool__` method, and thus control the path executed by the target program, as described below.

5.3 The Model Checking Loop

Before discussing how boolean proxies control execution of the target program, we first present the overall model checking loop of the `test` function in Figure 6, since the two are tightly coupled.

The `test` function maintains two global variables to record the model checking state:

- `__path__` is a list of booleans describing the branch chosen at each free branch point in the current path; this list is initialized to empty when `test` is first called and is updated each time a formula is evaluated;
- `__pathcondition__` contains the set of path constraints to be supplied to the SMT solver, which is expanded at each decision point with the selected constraint.

To symbolically execute a function we call the `test` method (line 6 of Figure 6) providing the target function `f` along with instances of the symbolic variables that the function takes as arguments. At each iteration `test` resets `__pathcondition__` to empty and executes the target function, tracking and reporting possible exceptions.

After the target function returns, the test function selects the next path (on line 15). Since symbolic execution always chooses to follow the true branch first and then the false branch, all the decisions points for which the false branch has been selected (which have already been explored) are popped from `__path__`. The code then checks if `__path__` is empty (which signals that the whole branch tree has been explored) in which case the testing is completed. If `__path__` is not empty, the code switches the last path decision to false, so that it can explore a new branch on the next iteration.

Additionally PEERCHECK can use the information stored in `__path__` and `__pathcondition__` to produce DOT graphs. These graphs allows us to visualize the paths that get chosen during symbolic execution.

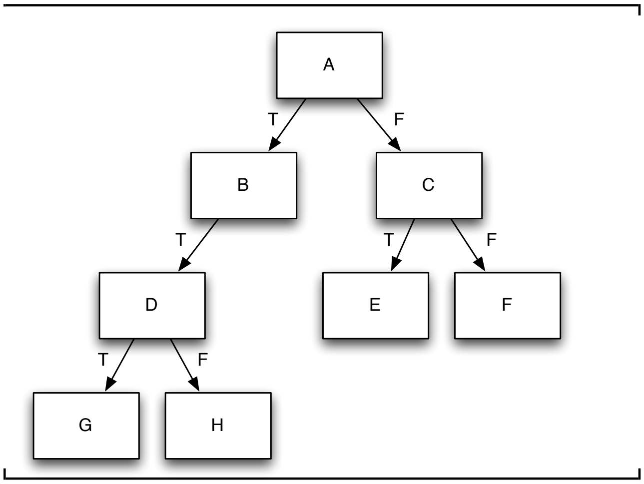
5.4 Boolean Proxies

At any conditional statement whose value depends on a symbolic input, the `__bool__` method of the `BoolProxy` class

Figure 6: Symbolic Execution Engine: Boolean Proxies and Model Checking

```
__path__ = []
__pathcondition__ = []

class DepthException(Exception): pass

def test(f, *args, **kwargs):
  global __path__, __pathcondition__
  __path__ = []
  while True:
    __pathcondition__ = []
    try:
      # Execute the target function
      result = f(*args, **kwargs)
    except Exception, e: print e

    # Set the next exploration path:
    # 1. remove all False branch points since they have been totally explored
    while len(__path__) > 0 and not __path__[-1]: __path__.pop()
    # 2. if path is empty the whole branch tree has been explored
    if __path__ == []: return
    # 3. switch the last true branch to false to explore the other path
    __path__[-1] = False

class BoolProxy(object):
  def __init__(self, formula): self.formula = formula

  def __not__(self): return BoolProxy( smt_fop('!', self.formula) )

  def __bool__(self):
    global __path__, __pathcondition__
    # Check if the true and the false branches can be selected
    true_cond  = smt_solve(smt_fop('&', __pathcondition__ + [self.formula]))
    false_cond = smt_solve(smt_fop('&', __pathcondition__ + [smt_pred('!', self.formula)]))

    # If just one branch can be selected, choose it leaving __path__
    # and __pathcondition__ unmodified
    if true_cond  and not false_cond: return True
    if false_cond and not true_cond: return False

    if len(__path__) > len(__pathcondition__):
      # The path has been decided by the test function, follow it adding
      # the relative constraint to __pathcondition__
      branch = __path__[len(__pathcondition__)]
      __pathcondition__.append(self.formula if branch else smt_pred('!', self.formula))
      return branch

    # If len(__path__) >= 10 the depth limit is reached, prune the search
    if len(__path__) >= 10: raise DepthException('Depth exceeded')

    # Follow the true branch and add __pathcondition__ with the new constraint
    __path__.append(True)
    __pathcondition__.append(self.formula)
    return True
```

Figure 7: Path Constraints



is called by the Python interpreter to decide which branch to execute. This `__bool__` method behaves as follows:

1. The SMT interface is used to check if both branches are feasible, that is, if the formula contained in the `BoolProxy`, and its negation, are both satisfiable when conjoined with the existing path condition.
2. If just one of the two branches are feasible, then the execution follows that branch by returning the appropriate boolean value, without increasing the size of the path or the path condition.
3. Otherwise, if the `__path__` array contains additional entries representing a desired path, then that path is chosen, and the path condition is extended appropriately.
4. Otherwise, if the depth bound is exceeded, then this path is terminated.
5. Otherwise, the `True` branch is explored on this test run, and the path condition is extended appropriately. The `False` branch is deferred to a subsequent run.

5.5 Additional Values and Theories

Although our idealized presentation handles just integers and booleans, our actual implementation uses the Z3 SMT solver, and we are able to take advantage of Z3 modules that have been developed to resolve many different types of constraints—like integer equations, strings and arrays—efficiently.

Integers Z3 can directly represent all the numeric data type operations required from the python standard interface [18].

Strings There are a number of Z3 plugins [4,19,20] that can be used to solve string constraints. String constraints are essential for analyzing programs in dynamic languages such as Python since they are often used in web development and system administration where strings are heavily used.

Arrays Array constraints are also supported by Z3. However the need to solve array constraints is mitigated by the ability of PEERCHECK to execute concrete arrays containing proxy objects.

Objects Objects in Python present a challenge not usually found in symbolic execution engines built in static languages. In static languages the structure of an object is defined before it is instantiated making symbolic execution straightforward. In Python however the structure of an object is not known initially so it must be inferred at runtime. This can be done by using proxy objects to record calls to methods and fields.

6. Contract system

To discover actual bugs in code, we need a way to encode the desired properties we want to assert about our executions. We developed a contract system similar to [11] but making use of the dynamic capabilities of python and the anonymous lambda functions to assert code properties.

The contract system, following the philosophy of the symbolic execution framework, is a *short* python library composed of only 100 lines of code. It allows us to assert preconditions and postconditions on functions and class methods.

Our contracts make use of the Python's decorator syntax to wrap methods and classes:

- `@inv(condition)` can be attached to any class and checks that `condition` is true at the end of object construction (after `__init__` runs) and after each method invocation;
- `@pre(condition)` checks that `condition` is true before the method invocation;
- `@post(condition)` checks that `condition` is true after the method invocation;
- `@arg(condition)` asserts that some property holds on the arguments.

Section 7 shows how this system, along with symbolic execution, is able to directly detect defects in the code with precision. Here it suffices to say that we just ignore precondition and argument exceptions, since they represent erroneous inputs. Instead we treat postcondition and class invariant errors as revelations of bugs in the checked code: the algorithm then, in a concolic execution fashion, retrieves a set of valid values for the symbolic variables from the SMT solver, and re-executes the code with these real values to produce the actual behavior.

7. Experimental Results

We tested the symbolic executor on the following example benchmarks:

- Quick Sort,

Figure 8: QuickSort Benchmark

```
@ensure(lambda result: ordered(result))
def quick_sort(array):
  if len(array) <= 1:
    return array
  pivot = array[0]
  less = []
  greater = []
  for x in array[1:]:
    if x <= pivot:
      less.append(x)
    else:
      greater.append(x)
  return quick_sort(less) + [pivot]
        + quick_sort(greater)
```

- Bubble Sort,
- the factorial function,
- an RB-tree.

For each of these examples we tested correctness properties of the implementation using the symbolic executor together with the contract framework. We then inserted some common bugs to validate that the framework can detect the discrepancy between the code's behavior and the desired properties.

7.1 QuickSort

As shown in Figure 8 we implemented a correct version of quicksort and tested it with an array of three symbolic variables $[x, y, z]$ to see all the six possible execution paths (corresponding to the six possible permutations for an array of three variables). The graph in Figure 11(a) produced by PEERCHECK shows each symbolic test node, labelled with the performed test and with two arrows for each possible result. We can also see from the graph that the execution path is forced from a certain point of each path, corresponding to the checks made by the contract system. The `ordered` postcondition checks that the resulting array is (non-strictly) increasing.

7.2 Factorial algorithm

The example in Figure 9 shows a simple yet purposely faulty implementation for computing a factorial. This shows that it is possible to go to a high depth of branch choices, provided most of the branches are free rather than forced. For example, even a depth limit of 2 free branches could detect the bug, since after the first test `x==40` evaluates to true, all the other tests performed in the contract are forced.

Since the factorial is computed correctly for all values except 40 and this is done through a specific test that adds the constraint $x = 40$ to the solver, all the other tests are forced. Therefore, we can safely go depthwise in our search avoiding the state space explosion. Our default limit of 10 free branches allows us to explore up to 2^{10} different paths, which is a reasonable finite amount to test procedures with loops. Since this algorithm stops once the test $x > 0$ fails, the symbolic executor produces just 10 different paths.

Figure 9: Factorial Benchmark

```
def fact_spec(x):
  n = 1
  while x > 0:
    n *= x
    x -= 1
  return n

@ensure(lambda x, res: fact_spec(x) == res)
def faulty_fact(x):
  if x == 40:
    return 123456789
  n = 1
  while x > 0:
    n *= x
    x -= 1
  return n
```

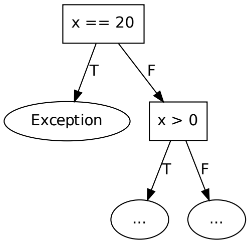


Figure 10. Factorial graph

7.3 RB-trees and class invariants

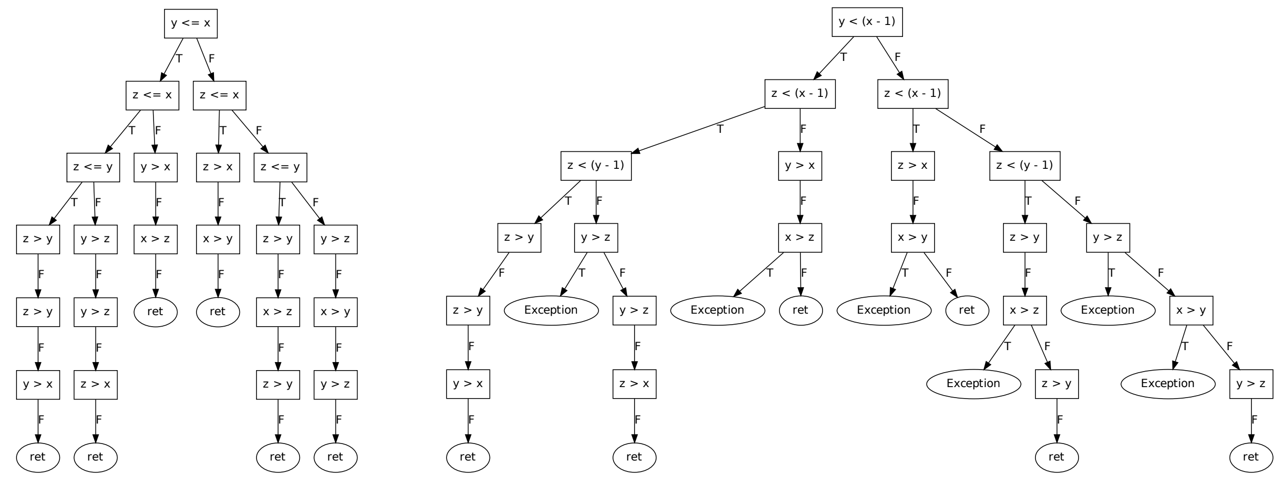
To test our contract system we modified an implementation of red black trees [13] to use our contracts.

We provided a test function that calls the insert and search methods for a set of provided variables, and we studied how the code coverage grows with the input size, as we can see in the following table:

Input Size	Run Statements	Percent
1	64	51%
2	73	58%
3	113	90%
4	121	97%
5	122	98%
6	123	98%
7	123	98%
8	125	100%

Red black trees are notably a delicate data structure, and finding an example input that explores all the possible cases is generally non trivial and requires a lot of thinking and deep understanding of the algorithm. Here we see that, ap-

Figure 11: QuickSort Execution Trees



(a) Correct QuickSort (b) Faulty QuickSort

Figure 12: Performance Test

```python
def guess(x):
  import random
  for i in range(0, 1000000):
    random.randint(0, 42)
  if x == random.randint(0, 42):
    print "Good guess"
```

plying symbolic execution, we reach 97% of code coverage supplying 4 variables, going up to 100% with 8 variables and a slightly increased depth limit of 13 steps.

7.4 Performance test

In Figure 12 we present an example that shows how our symbolic execution does not affect performance when the target program is loosely dependent on its parameters. This examples computes $1,000,000$ random numbers and then tests if `x` is equal to the next random number. Symbolically executing this procedure results in executing it two times, one where `x` is equal to the next random integer and one where it is not. Most of the computational time will be spent in generating the first $1,000,000$ random numbers.

The average time (over five runs) to execute the function twice with concrete values was 6.398, while the average time (also over five runs) to symbolically execute this procedure was 6.146 seconds. This shows that the overhead of the symbolic execution infrastructure is small when only a small part of the system is executed symbolically.

8. Related Work

Symbolic execution was first studied by King [8] who built a programming environment called EFFIGY that allowed symbolic interpretation of the programs written in its language.

In recent years there has been a resurgence of interest in symbolic execution. Meudec [9] use constraint logic programming and symbolic execution to automatically generate test data. Balser et al. [2] use symbolic execution to interactively prove properties of concurrent systems. Khurshid et al. [7] instrument Java code in Java Pathfinder to combine symbolic execution with model checking to combat the state space explosion problem of model checking. Pasareanu and Visser [10] develop a method of finding and proving loop invariants in Java code using symbolic execution. Berdine et al. [3] use separation logic and symbolic execution to automatically prove Hoare triples.

There has also been interest in using symbolic execution to automatically generate tests. Xie et al. [21] present a framework called Symstra that generates object-oriented unit tests using symbolic execution. Tillmann and Schulte [17] describe parameterized unit tests combined with symbolic execution to drive automated testing.

Sen et al. [15] studied and developed the Concolic Unit Testing Engine (CUTE) for C programs, which added a concrete execution step after the symbolic analysis process; the target program is executed with concrete values obtained from symbolic execution techniques.

Godefroid et al. [6] used symbolic execution to automatically generate test suites for C programs that provided high path coverage.

Anand et al. [1] add symbolic execution capabilities to the Java Pathfinder model checking tool. They also studied and developed a way to symbolically interpret data structures such as Java classes through lazy instantiation of references. Pasareanu et al. [12] extend the work done in [1] but rather than instrument code their system is a "nonstandard" bytecode interpreter. Their system uses the system-level concreate executions to improve unit test generation.

Similarly to the work done for Java Pathfinder, Tillmann and Halleux [16] implement a symbolic execution engine for the .NET framework called Pex. They also use Z3 to solve path constraints.

The main contribution of PEERCHECK is that, while each of these projects provide a complete running environment for the host language different from the compiler itself, our approach simply provides a lightweight symbolic execution library that can be plugged into existing python project to test modules.

References

[1] S. Anand, C. Păsăreanu, and W. Visser. JPF–SE: A Symbolic Execution Extension to Java PathFinder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, 2007.

[2] Michael Balser, Christoph Duelli, Wolfgang Reif, and Gerhard Schellhorn. Verifying Concurrent Systems with Symbolic Execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[3] Josh Berdine, Cristiano Calcagno, and Peter OHearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer Berlin / Heidelberg, 2005.

[4] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. 2009.

[5] L. De Moura 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*, pages 337–340. Springer-Verlag, 2008.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223. ACM, 2005.

[7] Sarfraz Khurshid, Corina Psreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer Berlin / Heidelberg, 2003.

[8] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[9] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, 2001.

[10] C.S. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. *Model Checking Software*, pages 164–181, 2004.

[11] R. Plosch. Design by contract for python. In *Software Engineering Conference, 1997. Asia Pacific ... and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings*, pages 213–219, December 1997.

[12] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[13] John Reid. Recipe 576817: Red black tree. `http://code.activestate.com/recipes/576817-red-black-tree/`, accessed February 2011.

[14] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck. In *Proceedings ACM SIGPLAN 2008 Haskell Workshop*, pages 37–48, 2008.

[15] K. Sen, 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*, pages 263–272. ACM, 2005.

[16] N. Tillmann and J. De Halleux. Pex: white box test generation for. NET. In *Proceedings of the 2nd international conference on Tests and proofs*, pages 134–153. Springer-Verlag, 2008.

[17] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE software*, pages 38–47, 2006.

[18] Guido van Rossum. Data Model — Python Documentation. `http://docs.python.org/py3k/reference/datamodel.html`, accessed February 2011.

[19] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 498–507. IEEE, 2010.

[20] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL Query Explorer. 2010.

[21] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer Berlin / Heidelberg, 2005.

[22] M. Young and R.N. Taylor. Combining static concurrency analysis with symbolic execution. *Software Engineering, IEEE Transactions on*, 14(10):1499 –1511, October 1988.