Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Точилина Екатерина Николаевна

Выпускная квалификационная работа

Динамическое символьное исполнение для языка Python

Уровень образования: бакалавриат
Направление 02.03.01 «Математика и компьютерные науки»
Основная образовательная программа СВ.5152.2020
«Математика, алгоритмы и анализ данных»

Научный руководитель:
доцент, Кафедра системного программирования,
к.ф.-м.н. Д. А. Мордвинов

Рецензент:
профессор,
Институт прикладных компьютерных наук,
университет ИТМО, к.т.н. В. М. Ицыксон

Санкт-Петербург
2024 г.

Содержание

Введение

Тема данной работы возникла в контексте задачи автоматической генерации юнит-тестов. Подобные инструменты способствуют улучшению качества программного кода и ускорению процесса тестирования. Автоматически создаваемые тесты позволяют выявлять ошибки и дефекты, обеспечивая более полное покрытие функциональности программы. Кроме того, они ускоряют обнаружение проблем при внесении изменений в код, что упрощает процесс рефакторинга. Поэтому разработка подобных инструментов является важной задачей.

Существуют различные подходы к генерации юнит-тестов [1]. Методы «черного ящика» состоят в том, что анализ тестируемой программы не производится. Есть подходы, когда из анализируемой программы извлекается небольшой набор данных, например, используемые константы. Такие подходы называют методами «серого ящика». Существуют методы «белого ящика», которые состоят в том, что собирается полная информация обо всех операциях внутри тестируемой программы. За счет проведения более полного анализа программы возможно достигать большего покрытия.

Один из распространенных методов для генерации тестов — фаззинг [2]. Он заключается в подстановке случайных данных в качестве входа функции, возможно, с использованием эвристик. Это метод «черного» или «серого» ящика. В определенных ситуациях фаззингу сложно пробиться через множество проверок входных данных внутри функции, что приводит к недостаточному покрытию. В таких случаях более эффективным методом является символьное исполнение [3], который является методом «белого ящика».

Метод символьного исполнения заключается в реализации интерпретатора, оперирующем символьными выражениями, с которыми умеют работать SMT-решатели[1]. Такой интерпретатор исследует различные пути выполнения программы и формирует ограничения, соответствующие этим путям. Затем SMT-решатель находит конкретные значения, удовлетворяющие собранным ограничениям.

[1] SMT (satisfiability modulo theories) — это задача разрешимости для логических формул с учётом лежащих в их основе теорий. Примеры таких теорий: теории целых и вещественных чисел, теории списков, битовых векторов и т. п.

Разработка анализаторов для Python является весьма сложной задачей. Это связано с разными аспектами данного языка программирования. Во-первых, в языке Python очень большое количество разных синтаксических конструкций, которые надо учитывать. Во-вторых, значительная часть стандартной библиотеки Python — это нативные расширения. К тому же есть много внешних библиотек, которые являются нативными расширениями, например, библиотека numpy.

Главная особенность Python заключается в том, что это динамически типизированный язык. Это представляет большие трудности как для фаззинга, так и для символьного исполнения. Неудачное определение типов входных данных может привести к быстрому завершению программы. При этом вывод типов в Python является очень сложной задачей сам по себе. В части случаев возможно учитывать информацию из аннотаций типов, однако они используются далеко не во всех проектах на Python, и часто являются неполными и неточными. Существующие инструменты справляются с данной проблемой недостаточно хорошо.

В данной работе был предложен подход, как можно моделировать объекты неизвестного типа в символьном виде, а также каким образом приоритизировать конкретные типы. Основная идея состоит в том, что в символьной машине вводится техника отложенных ветвлений.

Предложенный подход был реализован в рамках символьной машины USVM. Эксперименты показали, что данный подход позволяет генерировать тесты с покрытием строк на 24% больше, чем у лучшего из существующих инструментов символьного исполнения для языка Python. Также добавление техники символьного исполнения в инструмент для генерации юнит-тестов UnitTestBot позволило увеличить покрытие сгенерированных тестов на 8%.

Постановка задачи

Целью данной работы является разработка системы динамического символьного исполнения для языка Python, которую возможно использовать для улучшения качества автоматической генерации юнит-тестов.

Для достижения поставленной цели были выделены следующие задачи:

- Анализ существующих реализаций символьного исполнения для языка Python.
- Разработка подходов для решения проблем символьного исполнения, специфичных для языка Python.
- Реализация разработанных подходов.
- Экспериментальное сравнение итогового инструмента с существующими аналогами.

1. Обзор существующих решений

1.1. Инструмент CrossHair

На данный момент инструмент CrossHair [4] — это наиболее полная реализация символьного исполнения для Python.

Основная задача данного инструмента заключается в том, чтобы проверять описанные программистом инварианты функций и находить контрпримеры, где они нарушаются. В инструменте CrossHair также присутствует подпрограмма cover, которая призвана покрыть функцию тестами, что ближе к нашей задаче. Однако генерация кода юнит-тестов — это отдельная большая задача, и в CrossHair она выполнена недостаточно хорошо: в определенных ситуациях генерируются тесты, в которых неправильный синтаксис. Такое происходит из-за того, что рендеринг объектов реализован наивным образом, который работает лишь для очень ограниченного набора типов (int, str, bool и т. п.).

1.1.1 Проблема нативных функций

Инструмент CrossHair реализует подход из статьи [5]. Он заключается в следующем: символьная машина представляет собой библиотеку на Python, в которой реализованы символьные аналоги стандартных типов. Они моделируют поведение соответствующих типов на примитивах SMT-решателя. В статье представителей этих аналогов называют прокси-объектами. Для символьного анализа они подаются на вход оригинальной анализируемой функции. Это возможно, так как в Python «утиная» типизация. На конкретной итерации прокси-объекты ведут исполнение в соответствии с конкретными значениями, отлавливают ветвления, генерируют значения, соответствующие другим веткам, и на следующих итерациях ведут исполнение по другому пути.

Значительный недостаток данного подхода в том, что корректно работать с прокси-объектами могут лишь функции, написанные на Python. А огромное количество функций в стандартной библиотеке — нативные. Чтобы они корректно работали с прокси-объектами, их необходимо заменить собственными реализациями. При пропуске разработчиками некоторых натив-

ных функций, мы получим некорректное исполнение в символьном анализе. Та же проблема возникнет в ситуации, когда анализируемый код использует нестандартное нативное расширение.

Ниже представлены примеры, где проявляется описанная выше проблема подхода.

1. Пример с использованием внешней нативной библиотеки (здесь это популярная математическая библиотека numpy):

Анализируемый код	Вывод crosshair.cover	Реальное поведение при подстановке найденных значений

`import numpy as np`
 	
`def f(x: int):`
`  y = np.array(x)`
`  return y.dtype`
	`def test_f():`
	`  assert f(0) == dtype('O')`
		`>>> f(0)`
		`dtype('int64')`

Таблица 1: Исходя из вывода функции crosshair.cover, мы понимаем, что в его исполнении данная функция возвращает dtype('O'), что соответствует numpy-массиву из сложных объектов. Однако если мы запустим функцию на нуле «чистыми» образом, окажется, что реально функция возвращает dtype('int64').

2. Пример нереализованного аналога нативной функции из стандартной библиотеки (здесь используется функция для сериализации произвольных объектов pickle.dumps):

1.1.2 Проблема неизвестных типов

В Python коде часто возникает ситуация, когда параметры функций не имеют аннотаций типов. Тогда CrossHair подставляет некоторые простые стандартные типы, однако не всегда они оказываются ожидаемыми.

Пример функции:

Анализируемый код	Вывод crosshair.cover	Реальное поведение при подстановке найденных значений

`import pickle`
 	
`def f(x: int):`
`  return \`
`  pickle.dumps(x)`
	`import pytest`
	
	`def test_f():`
	`  with pytest.raises(`
	`  ValueError):`
	`  f(0)`
		`>>> f(0)`
		b ' \x80\x04 K \x00 . '

Таблица 2: Когда нативная функция pickle.dumps получает на вход прокси-объект, ожидаемо, что сериализовать она его не сможет, что мы и видим по выводу функции crosshair.cover. Однако очевидно, что настоящий ноль сериализовать возможно, поэтому получившееся исполнение не корректно.

```
def floy(a_and_n):
    (a, n) = a_and_n
    dist = list(a)
    path = [[0] * n for i in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
                path[i][k] = k
    print(dist)
```

Анализируя код, можно сделать вывод, что функция ожидает входные данные в формате tuple[list[list[int]], int] или аналогичном. Однако CrossHair подставляет только строку, за счет чего покрытие тестов маленькое:

```
import pytest
from main import floy
def test_floy():
    with pytest.raises(TypeError):
       floy('\x00\x00')
```

1.2. Инструмент PyExZ3

Проект PyExZ3 [6] также реализует идею из статьи [5], поэтому здесь тоже присутствуют проблемы с нативными функциями. В случае отсутствия указанных типов подставляется int, то есть вывод типов отсутствует. Большой недостаток данного инструмента в том, что он сделан для старой версии Python и больше не поддерживается: последнее обновление было в 2015 году.

1.3. Инструмент PySym

Как отмечает автор инструмента PySym [7], это скорее учебный скрипт, который может быть полезен другим людям, чем готовый к использованию анализатор кода. На данный момент поддержаны символьные операции только для примитивных типов. Проект не обновляется с 2019 года.

1.4. Генераторы тестов без символьного исполнения

1.4.1 Инструмент Pynguin

На данный момент самый популярный генератор юнит-тестов для языка Python — это Pynguin [8]. Этот инструмент использует фаззинг, особенность которого в том, что он генерирует цепочки вызовов разных функций одного модуля вместо того, чтобы фиксироваться на одной конкретной функции. Аналогом этого инструмента для языка Java является инструмент EvoSuite [9], который регулярно побеждает на соревнованиях по генерации юнит-тестов для языка Java [10].

Инструмент Pynguin активно поддерживается и развивается.

1.4.2 Инструмент Klara

Klara [11] — инструмент статического анализа и автоматической генерации юнит-тестов, основанный на анализе абстрактного синтаксического дерева. Внутри себя он также использует SMT-решатель, однако это не инструмент символьного исполнения.

Данный инструмент не поддерживает большое количество синтаксических конструкций, в частности, циклы. Из-за этого использовать его на практике практически невозможно. Список ограничений данного инструмента представлен в его документации [12]. Проект не обновляется с 2021 года.

1.4.3 Инструмент Hypothesis

Hypothesis [13] — это популярная и развивающаяся библиотека для тестирования на основе свойств (Property-Based Testing) для Python. Это не генератор юнит-тестов в чистом виде, однако в нем есть подпрограмма Ghostwriter [14], которая автоматически генерирует тесты с использованием исходной библиотеки.

В данный инструмент встроен фаззинг, который воспринимает тестируемую программу как «черный ящик», не учитывая внутреннее устройство. Из-за этого Hypothesis может не найти проблемы в коде, которые происходят на некоторых очень конкретных данных. Достигаемое им покрытие, как правило, невелико по сравнению с инструментами, проводящими дополнительный анализ тестируемого кода.

2. Описание алгоритма

2.1. Выбранный подход

Реализация символьного исполнения предполагает написание своего интерпретатора языка на символьных примитивах, с которыми умеют работать SMT-решатели. Этот интерпретатор может работать самостоятельно, тогда мы имеем статический анализатор кода, а можно комбинировать работу символьного интерпретатора с оригинальным.

Первый вариант выигрывает по скорости и возможностям использования разных техник символьного исполнения. Однако в случае языка Python из-за богатства встроенных конструкций полная реализация интерпретатора является очень сложной задачей. Если у нас есть символьный интерпретатор языка Python, его в частности можно использовать как альтернативный конкретный интерпретатор, а с альтернативными реализациями языка Python все непросто.

Официальный интерпретатор языка Python — CPython. На одной из страниц его сайта [15] собраны его альтернативы. Отмечается, что из всех рассмотренных интерпретаторов совместимы с некоторыми версиями CPython только три инструмента: PyPy [16], Jython [17] и IronPython [18]. Из них наиболее активно поддерживаемый интерпретатор — PyPy, но на данный момент в нем есть поддержка только версий 2.7, 3.9, 3.10, в то время как последняя версия CPython — 3.13. Jython поддерживает только 2.7, а IronPython — 2.7 и 3.4. Большая проблема поддержки альтернативных интерпретаторов в том, что с каждой новой версией сильно меняется байт-код.

Поэтому, полностью поддержать все инструкции языка — это трудоемкая задача. Когда наш интерпретатор будет сталкиваться с неизвестными инструкциями, статический анализ будет остановлен.

В подобных ситуациях можно обеспечивать более мягкое поведение с помощью оригинального интерпретатора. Связывать работу конкретного и символьного интерпретаторов можно различными способами. Существующие реализации символьного исполнения для Python используют подход из статьи [5], но он имеет принципиальные недостатки, которые были разобраны

в предыдущем разделе на примере CrossHair.

Было решено пойти по пути модификации исходного кода CPython. Модификация заключается в том, что теперь интерпретатор умеет работать с обертками над объектами, в которых содержится информация как о его конкретном представлении, так и о символьном. При проведении неизвестных операций информация о символьном представлении объекта теряется, но исполнение не останавливается. При проведении известных операций с объектом вызывается реализация этой операции в символьном виде. Модифицированный CPython предоставляет API для произвольной реализации известных символьных операций.

Проиллюстрируем данный подход на примере, изображенном на рис. 1.

Здесь анализируется функция взятия модуля у целого числа my_abs. На указанных шагах происходят следующие действия:

1. Создается обертка из случайного конкретного числа (здесь это 5) и численного символа без ограничений. Эта обертка подается на вход функции.

2. Происходит операция сравнения с нулем. Эта операция разделяется на 2 части — конкретную и символьную. Конкретная версия производится в CPython, символьная — в символьной машине.

   Мы также сообщаем символьной машине, что по результату этой операции происходит ветвление. В данном случае конкретное значение — true, поэтому в следующий раз надо будет подобрать входные данные таким образом, чтобы получилось false.

3. Завершаем итерацию: получился тест, соответствующий первой ветви исполнения.

4. Запускаем функцию my_abs на новой обертке. Теперь значение x отрицательное.

5. Завершаем итерацию: получился тест, соответствующий второй ветви исполнения.

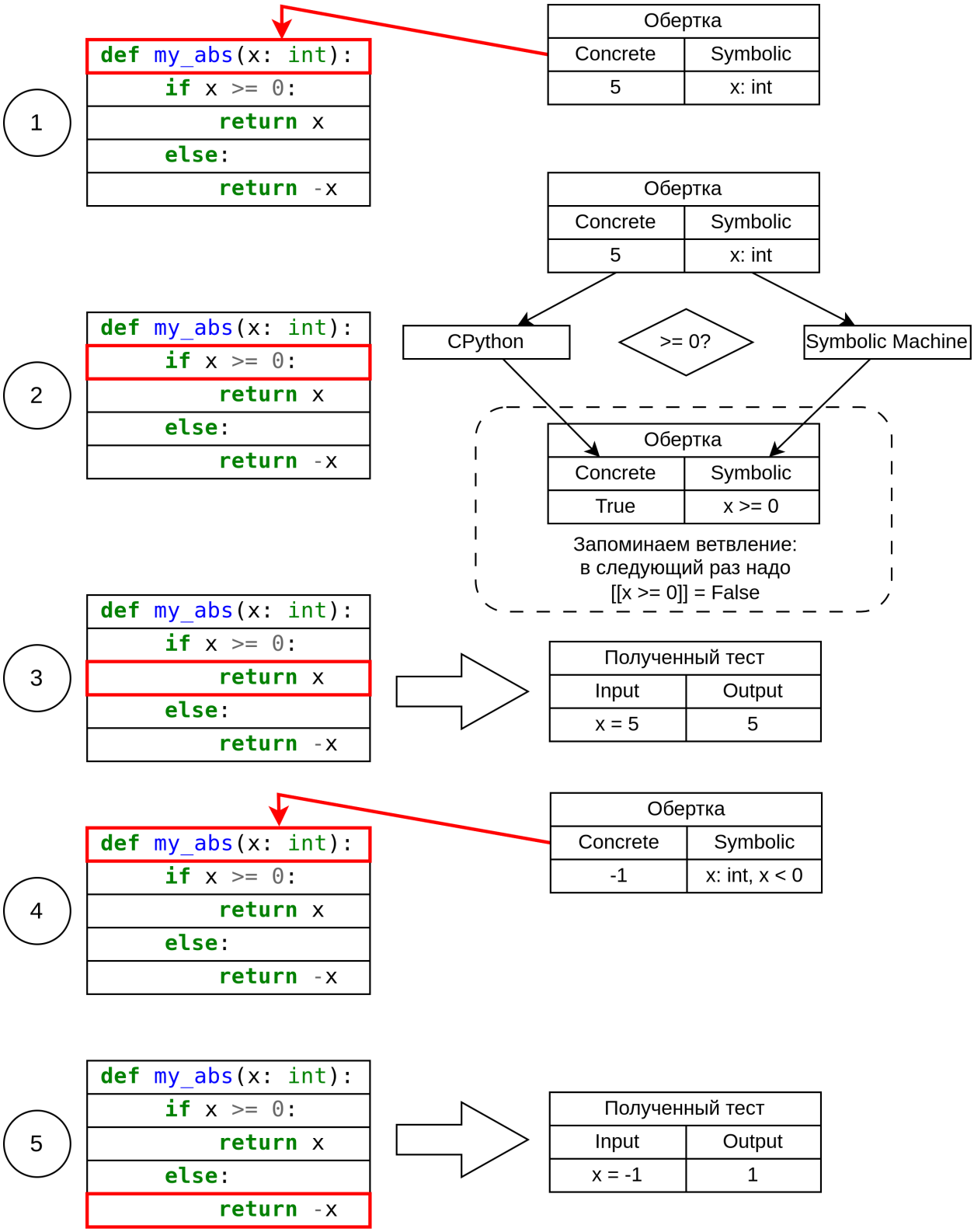


Рис. 1: Пример совместного конкретного и символьного исполнения

2.2. Компоненты универсальной символьной машины

Символьные анализаторы разрабатываются для разных языков программирования, и во многом решаемые там задачи пересекаются. В данной работе используется проект USVM [19] — универсальная символьная виртуальная машина. В ней уже решено большое количество задач, например:

- Настроено взаимодействие с несколькими SMT-решателями за счет использования библиотеки KSMT [20].
- Разработана модель символьной памяти, благодаря чему интерпретаторы разных языков работают с достаточно высокоуровневыми абстракциями (символьными и конкретными адресами, полями объектов).
- Реализованы символьные модели стандартных контейнеров: массивов, словарей, множеств.
- Реализовано несколько алгоритмов выбора следующего символьного состояния.
- Есть поддержка решения типовых ограничений на объекты.
- Использованы разные оптимизации, которые позволяют уменьшить количество обращений к SMT-решателю.

В данной работе решаются проблемы, которые специфичны для языка Python. Более общие вопросы здесь не затрагиваются, используются уже готовые решения.

2.3. Основной цикл символьного исполнения

Символьное состояние — это абстракция, в которой хранится информация о пройденном пути, символьной памяти, собранных логических ограничениях. При посещении конструкций ветвления происходит порождение новых состояний, которые затем попадают в очередь. Цикл повторяется до тех пор, пока в очереди есть состояния, или пока анализ не будет остановлен по каким-то другим причинам.

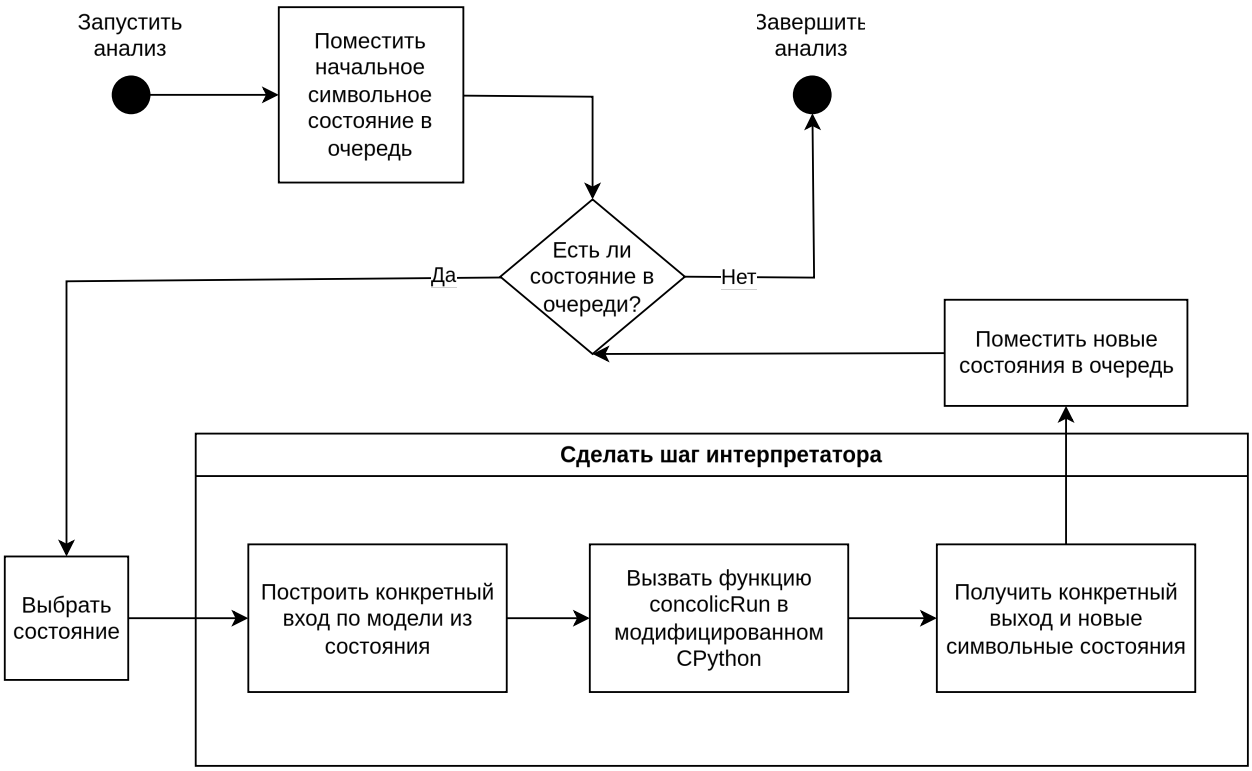


Рис. 2: Основной цикл символьного исполнения

В нашей реализации в каждом символьном состоянии также есть ровно одна модель, которая удовлетворяет всем ограничениям. Здесь под моделью подразумевается отображение всех возможных символьных значений на конкретные. Когда порождается новое символьное состояние, необходимо породить новую модель. В этот момент происходит вызов SMT-решателя. Вся логика перевода конструкций решателя в структуры символьной машины и обратно уже реализована в ядре USVM.

По модели можно построить настоящий Python объект. Для каждого аргумента собирается пара из конкретного объекта и его символьного представления, и на них запускается исполнение модифицированного CPython.

2.4. Работа с неизвестными типами

Одна из принципиальных задач, которая возникает при разработке символьной машины для Python, — это организация работы с объектами неизвестных типов.

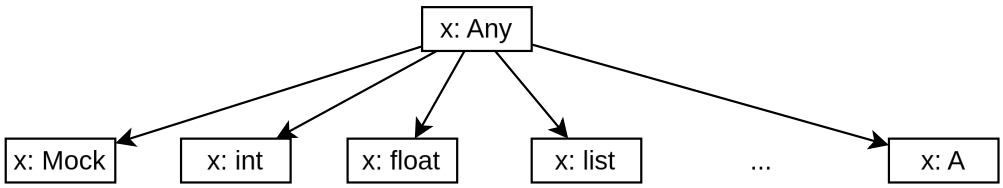
Рассмотрим для примера следующую программу на языке Python.

```
class A:
    def __add__(self, other) -> A:
        ...
    def run(self):
        ...

def f(x):
    y = x + x
    y.run()
```

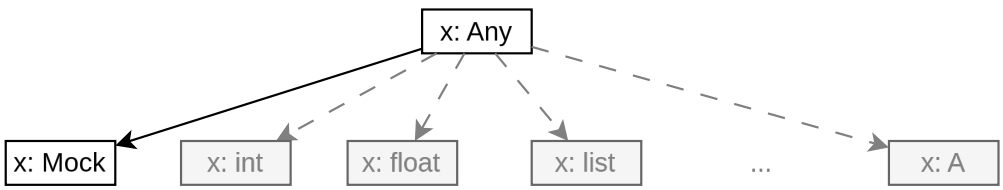
Задача состоит в том, чтобы символьно исполнить функцию f.

Операция сложения определена для каждого типа по-своему. Символьному интерпретатору необходимо зафиксировать исполняемый код, поэтому в наивном решении в момент вызова операции __add__ происходит ветвление по типу переменной x.



Первый указанный тип — Mock — означает специальный имитирующий объект, у которого определены все возможные операции.

В наивном подходе исполнение скорее всего даже не успеет дойти до интересного типа — A. Однако, если бы решение о предпочитаемом типе принималось не сразу, а только после исполнения с имитирующим объектом, то для выбора можно было бы использовать информацию о том, что в дальнейшем у объекта вызывается метод run(). Поэтому для поиска конкретного типа используется отложенное ветвление.



После завершения исполнения с имитирующим объектом была собрана

следующая информация:

$$x = a : \mathrm{Mock}$$
$$y = x + x = \mathrm{Mock}(a.\_\_\mathrm{add}\_\_(a))$$
$$y.\mathrm{run}() = (x + x).\mathrm{run}() = \mathrm{Mock}(\mathrm{Mock}(a.\_\_\mathrm{add}\_\_(a)).\mathrm{run}())$$

Исходя из этих соотношений, можно сгенерировать протокол для типа переменной x.

```
class ProtocolForX(typing.Protocol):
    def __add__(self, other) -> ProtocolForY: ...

class ProtocolForY(typing.Protocol):
    def run(self): ...
```

Затем возвращаемся к отложенному ветвлению и с помощью сгенерированного протокола выставляем приоритеты имеющимся типам: сначала идут те типы, которые протоколу удовлетворяют. В общем случае протоколов может быть сгенерировано несколько.

3. Реализация

3.1. Связывание интерпретатора CPython и USVM в одном процессе

Рассмотрим связку символьного и конкретного интерпретаторов более подробно.

CPython написан на языке C. USVM, на основе которого реализуется символьный анализатор для языка Python, — на Kotlin, соответственно основная часть машины тоже должна быть на Kotlin.

Символьный и конкретный интерпретатор должны работать в одном процессе, потому что в противном случае слишком много времени будет тратиться на межпроцессное взаимодействие. Соединять C и JVM в одном процессе можно с помощью технологии JNI — Java Native Interface.

С точки зрения CPython, объект — это произвольная структура на C, которая имеет префикс определенного вида. Объект однозначно определяется своим адресом. В машине со стороны JVM конкретный Python объект задается одним числом — его адресом. Взаимодействовать с таким объектом можно с помощью нативных вызовов, которые на стороне C вызывают разные функции из Python C API [21].

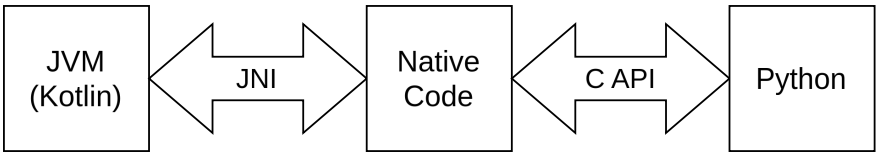


Рис. 3: Схема связывания Kotlin и Python в одном процессе

Процесс анализа запускается из JVM. В начале итерации по модели из символьного состояния строятся конкретные аргументы, и вместе со ссылками на их символьные представления они отдаются специальной функции concolicRun из модификации CPython. Когда возникает необходимость вызвать символьную реализацию той или иной операции, из нативного кода с помощью JNI вызывается соответствующая JVM-функция.

3.2. Модификация интерпретатора CPython

Модифицированный CPython отличается от оригинального тем, что везде, где можно, он старается работать с обертками объектов, которые содержат ссылки на обычный конкретный объект и его символьное представление.

Рассмотрим структуру Python объекта. В CPython это C структура данных, которая имеет префикс фиксированного вида — PyObject_Head. После него могут идти произвольные данные.

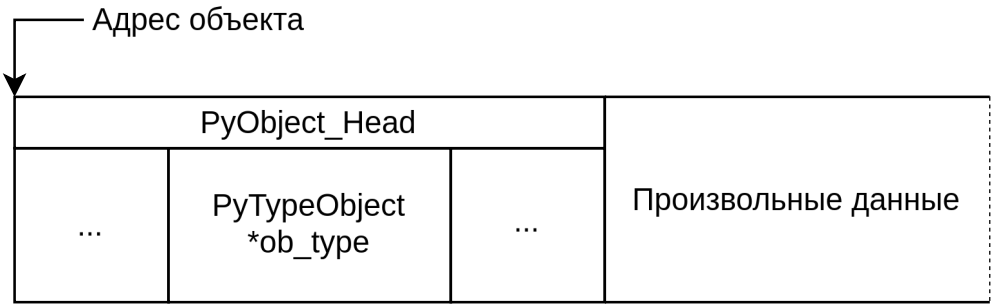


Рис. 4: Структура произвольного Python объекта

Интерпретатор понимает, как работать с таким объектом, благодаря ссылке на PyTypeObject, тип объекта, который в свою очередь также является Python объектом.

Тип объекта имеет определенную структуру [22]. Он описывает, что с объектом данного типа можно делать: какие у него определены операторы, как его создавать, как его должен удалять сборщик мусора.

Операций, которые можно производить над объектом, большое, но конечное число. Обертки, которые будут отдаваться интерпретатору вместо настоящих объектов, должны максимально имитировать поведение этих объектов, но при этом должна быть возможность отлавливать происходящие с ними операции.

Обертки реализованы в виде нативных Python объектов. Так мы имеем больше контроля, чем если бы обертки были реализованы на Python. Для каждого настоящего типа создается тип обертки, в котором определены в точности те же операции, что и у оригинального типа. Но реализации этих операций иные: они сначала вызывают нужную операцию для конкретного объекта, а потом вызывают соответствующую символьную операцию для его

символьного представления. В текущей реализации в этот момент происходит JNI-вызов в символьную машину.

Для примера ниже приведен псевдокод операции nb_negative, которая соответствует Python методу __neg__:

function wrapper_nb_negative($self$)
$concrete\_result \leftarrow nb\_negative(self.concrete)$
$symbolic\_result \leftarrow nb\_negative(self.symbolic)$
return $wrapper(concrete\_result,\ symbolic\_result)$
end function

Модификация CPython позволяет реализовать символьные варианты операций произвольным образом.

По умолчанию модифицированный CPython ведет себя как оригинальный. Режим работы с обертками запускается через функцию concolicRun, которой необходимо предоставить реализацию символьных операций, Python функцию, конкретные параметры и их символьные представления.

3.3. Реализация символьных операций

На данный момент поддержаны основные операции для следующих типов данных:

- Целые числа. Так как в языке Python поддержана длинная арифметика и переполнений не бывает, для моделирования символьных операций используется сорт Int из библиотеки KSMT, а не битовые вектора.

- Вещественные числа. Для наиболее точного моделирования стоило бы использовать числа с плавающей точкой из KSMT, но довольно быстро выяснилось, что в Z3[2] они работают слишком медленно. Было решено моделировать их через рациональные числа в KSMT, которые имеют произвольную точность. То, что в таком подходе могут возникать неточности, создает не очень большие проблемы. Во время исполнения мы все время сверяемся с конкретным интерпретатором. Если вдруг конкретный и символьный интерпретатор разойдутся, это событие будет

[2]SMT-решатель, который был выбран для символьного анализатора Python

отловлено и дальше просто перестанут производиться новые ветвления.

- Списки и кортежи. Реализация этих двух структур основывается на моделировании массивов в ядре USVM.

  Здесь отдельной большой проблемой было то, что SMT-решатель довольно часто подкидывал модель, в которой массив лежит в самом себе. Сейчас эта проблема решается некоторыми трюками, которые подправляют модель из решателя.

- Словари и множества. Реализация этих двух структур основывается на моделировании словарей и множеств в ядре USVM.

- Объекты со стандартной реализацией добавления и чтения полей.

  Поясним данную формулировку. В Python синтаксическая конструкция ”obj.field” компилируется в инструкцию байт-кода вида (LOAD_ATTR TOS ’field’), где TOS означает объект на вершине стека. Во время интерпретации эта инструкция берет тип объекта на вершине стека и вызывает его метод tp_getattro с параметрами obj и ’field’, если тот определен.

  Таким образом, в общем случае в момент взятия аттрибута у объекта может произойти произвольная операция. Тем не менее значительная часть типов имеет стандартную реализацию методов tp_getattro и tp_setattro. Сейчас работа с полями поддерживается только для объектов с такими типами.

  Стандартная реализация пары методов tp_getattro и tp_setattro работает с дескрипторами, записанными в типе, а также со словарем объекта, в котором содержится информация об индивидуальных аттрибутах. В символьной машине работа с такими аттрибутами реализована через словари из ядра USVM.

- Строки пока поддерживаются только константные, так как универсальную поддержку строк планируется добавить в ядре USVM позже.

Стоит заметить, что описанная выше схема с парами из конкретной и символьной реализаций операций применима не всегда. Например, нельзя реализовать в виде единой символьной операции лексикографическое сравнение списков. Для подобных случаев был разработан механизм добавления аппроксимаций, когда реальная нативная реализация операции заменяется на ее аналог, который написан на Python. Подмена происходит при прохождении операции через обертку в модифицированном CPython.

3.4. Система типов

Типовая система — важная компонента символьной машины для языка Python. Опишем ее устройство.

Сначала стоит уточнить, что в самом языке Python есть несколько «типовых систем». Во-первых, есть типы в интерпретаторе — ссылки на PyTypeObject. Внутри интерпретатора для этих типов есть отношение подтипирования. Однако, например, на уровне интерпретатора нет понятия дженериков — все контейнеры (списки, словари, множества) гетерогенны.

Во-вторых, есть опциональная статическая система типов, про которую интерпретатор ничего не знает. Она описана в PEP 483 [23] и PEP 484 [24]. Поддержкой этих типов занимаются отдельные статические анализаторы, самый популярный из которых — mypy [25]. В этой системе есть довольно сложные структуры — типовые переменные, дженерики, протоколы, рекурсивные типы. Для нас наиболее важными являются протоколы. Как они применяются, было описано в разделе 2.4.

Для работы с информацией о статических типах был написан прототип соответствующей библиотеки на Kotlin[3]. Для получения типовой информации на анализируемой программе запускается подправленный mypy, собранная им информация выкачивается в виде JSON, затем результат десериализуется в структуры на Kotlin. На этой стороне реализованы некоторые операции с типами, в том числе проверка на подтипирование.

Типовая система в нашей символьной машине — PyTypeSystem — не копирует ни динамическую, ни статическую систему типов языка Python.

[3] https://github.com/UnitTestBot/PythonTypesAPI

Она нужна для использования тех возможностей по работе с типами, которые предоставляет ядро USVM. Это наложение типовых ограничений на символы, решение этих ограничений и получение множества возможных типов в модели у объекта по конкретному адресу.

Было решено разбить типы в машине на 3 вида: конкретные, виртуальные и специальный тип Mock. Конкретный тип — тип, соответствующий объекту PyTypeObject. Виртуальный тип в общем случае — некоторый предикат для конкретных типов. Такие предикаты могут накладываться в разных ситуациях. Например, если на символе неизвестного типа интерпретатор позвал nb_negative, накладывается предикат, что в PyTypeObject поле nb_negative должно быть ненулевым.

Чтобы объяснить назначение типа Mock, вернемся к началу шага интерпретатора в основном цикле символьного исполнения, который описан в секции 2.3. Напомним, что для вызова функции concolicRun необходимо построить конкретные аргументы. Тут возникает вопрос, как конструировать конкретный объект, когда тип аргумента еще не определен.

В такой ситуации мы создаем имитирующий объект особого нативного типа, который невозможно сконструировать при обычном исполнении Python кода. В большинстве случаев при вызове методов такого объекта возвращается новый имитирующий объект. Однако есть особые методы, например nb_bool, которые не могут возвращать произвольный Python объект. Вызов nb_bool на имитирующем объекте соответствует ситуации ветвления по его значению. В этом случае в момент конкретного исполнения nb_bool происходит JNI-вызов в символьную машину, где значение true или false определяется исходя из текущей модели в символьном состоянии.

Опишем общую схему определения типа конкретного представления аргумента перед вызовом concolicRun. Ядро USVM позволяет запросить множество тех типов, которые объект по конкретному адресу в данной модели может иметь. Это множество представляется в виде абстракции UTypeStream.

В нашей машине поддерживается инвариант: UTypeStream или содержит единственный тип (конкретный или Mock), или содержит список типов, который начинается с Mock, а дальше состоит из подходящих конкретных типов. Таким образом, для построения конкретного аргумента выбирается

первый тип из выданного моделью UTypeStream.

3.5. Реализация отложенных ветвлений

Объяснение необходимости реализации отложенных ветвлений содержится в разделе 2.4. На данный момент это только ветвления по типу некоторого символа.

Компонента USVM, которая отвечает за выбор следующего символьного состояния, — это UPathSelector. Отложенные ветвления были реализованы в виде особого наследника этого класса — PyVirtualPathSelector. Этот наследник является «конструктором». Во-первых, внутри себя он содержит несколько очередей, которыми могут быть произвольные наследники UPathSelector. Во-вторых, внутренние операции в PyVirtualPathSelector подразумевают возможность использования разных стратегий.

Интерфейс UPathSelector выглядит следующим образом:

```
interface UPathSelector<State> {
    fun isEmpty(): Boolean
    fun peek(): State
    fun remove(state: State)
    fun add(states: Collection<State>)
    fun update(state: State)
}
```

Особенность PyVirtualPathSelector в том, что операция peek может сгенерировать новое символьное состояние из отложенного ветвления и только после этого отдать уже существующее.

Основные операции в PyVirtualPathSelector — это peek, add и remove. В нашем случае peek может возвращать null, но из такого варианта можно сделать пару функций isEmpty и peek, который null не выдает. С помощью кэша можно сделать так, чтобы несколько раз подряд вызванный peek выдавал одно и то же. Операция update выражается через последовательное применение remove и add. Опишем схему работы add, peek и remove.

Введем понятие дерева отложенных ветвлений. Каждой вершине этого

дерева, кроме корня, соответствует некоторое отложенное ветвление. Если отложенное ветвление является первым в символьном состоянии, в котором оно произошло, родителем соответствующей вершины является корень. Если отложенное ветвление не было первым, родителем его вершины является вершина предыдущего отложенного ветвления.

В PyVirtualPathSelector хранится дерево отложенных ветвлений. Каждое символьное состояние в нем соответствует вершине своего последнего отложенного ветвления или корню, если отложенных ветвлений не происходило.

Поймем, что должно храниться в каждой вершине. Здесь стоит вспомнить, ради чего все затевалось. Есть некоторый символ неизвестного типа, и у нас возникла необходимость этот тип конкретизировать. Вариантов слишком много, чтобы просто перебрать, поэтому надо расставить приоритеты. Проблема в том, что информацию о предстоящих операциях на момент ветвления мы еще не знаем. На «разведку» отправляется имитирующий объект, то есть состояние, в котором наш символ конкретизировался в тип Mock. Это состояние может дальше ветвиться и ходить по разным путям, в которых с символом будут происходить разные операции (в Python это относительно распространенная практика). Когда состояние-разведыватель завершило исполнение, мы можем извлечь из него информацию о том, какие методы вызывались у имитирующего объекта. Тогда по этой информации можно сгенерировать набор протоколов, который используется для выставления приоритетов.

Что мы в итоге имеем. В вершине отложенного ветвления должен храниться список типов, которые мы уже пробовали, а также набор рейтингов типов. Каждое состояние-разведыватель, завершив исполнение, добавляет в вершину свой рейтинг. То, к какому рейтингу прислушиваться, определяется в конкретной стратегии для PyVirtualPathSelector. Также в конкретной реализации можно хранить в вершине какую-то дополнительную информацию.

Из вершин дерева отложенных ветвлений есть отображение в множество очередей состояний (наследников UPathSelector). Это отображение задается в конкретной конфигурации PyVirtualPathSelector, но сейчас есть только одна реализация. В этом варианте 2 очереди: одно для состояний без отложенных ветвлений, другое — с ними. Соответственно, корень отображается в одну

очередь, остальные вершины — в другую.

При операции add новое состояние попадает в нужную вершину и соответствующую очередь. Если мы увидели отложенное ветвление в этом состоянии впервые, соответствующая вершина в дереве создается.

При вызове операции peek может произойти несколько событий. Выбор определяется стратегией конкретного представителя PyVirtualPathSelector.

function peek
  while true **do**
    $action \leftarrow$ выбрать действие по стратегии
    switch $action$ **do**
      case вариантов действий не осталось
        return $null$
      case выбрать состояние из $pathSelector$
        $state \leftarrow pathSelector.peek()$
        return $state$
      case выполнить отложенное ветвление в $vertex$
        $rating \leftarrow$ выбрать рейтинг в $vertex$ по стратегии
        $type \leftarrow$ выбрать тип из $rating$
        $state \leftarrow$ сгенерировать новое состояние
        $pathSelector \leftarrow$ взять из $vertex.parent$
        $pathSelector.add(state)$
  end while
end function

Варианты действий:

1. Можно отдать состояние, уже находящееся в одной из внутренних очередей.

2. Можно сгенерировать новое состояние из отложенного ветвления. Для этого выбирается вершина, в вершине выбирается рейтинг, и символ конкретизируется на первый еще не проверенный тип в этом рейтинге. Потом новое состояние отправляется в очередь, соответствующую родительской вершине. После данного шага состояние еще выбрано не

было, поэтому выбор необходимо повторить. Цикл продолжается, пока стратегия не решит отдать существующее состояние.

При вызове remove происходит удаление состояния из той внутренней очереди, в которой оно ранее лежало. Если состояние завершило исполнение, его можно использовать как состояние-разведыватель для совершенных отложенных ветвлений и сгенерировать в них новые рейтинги.

Есть еще случай, когда состояние исполнилось до конца, в нем не происходило отложенных ветвлений, тем не менее остались объекты неопределенного типа. Это означает, что вместо них можно поставить объекты любого конкретного типа из соответствующего UTypeStream. О таких состояниях в абстрактном PyVirtualPathSelector стоит сообщать, и уже в конкретных ситуациях разбираться, как с ними правильнее поступать.

3.6. Варианты реализаций PyVirtualPathSelector

В предыдущем разделе было указано, что PyVirtualPathSelector предполагает внутри себя реализацию некоторых стратегий для внутренних операций. В этом разделе рассмотрим некоторые из них.

3.6.1 Внутренние очереди

Первый аспект PyVirtualPathSelector, предполагающий вариативность, — это выбор конкретных реализаций внутренних очередей, а также способ отображения вершин дерева отложенных ветвлений на представителей данных очередей. В основном варианты реализации внутренних очередей рассматривались из уже готовых реализаций UPathSelector в ядре USVM.

Из способов отображения вершин на очереди был опробован лишь один вариант: одна очередь состояний для корня, одна — для всех остальных вершин. Разбиение состояний на 2 очереди необходимо потому, что стандартные реализации UPathSelector не могут учитывать информацию об отложенных ветвлениях, ведь на данный момент они бывают только в реализации для языка Python.

В качестве самой базовой очереди был опробован DfsPathSelector, который отдает последнее положенное в него состояние. Также из вариантов реа-

лизаций UPathSelector из ядра USVM был опробован RandomTreePathSelector, который на данный момент является одним из лучших в машинах для других языков.

Состояния можно приоритизировать на основе неких весов. Есть гипотеза, что довольно важным признаком является количество покрытых инструкций (без повторений). В языке Python при неизвестных типах генераторам тестов сложно пробиться через определенные инструкции, которые не завершают исполнение с ошибкой только для определенных типов данных. Если состояние смогло пробиться через что-то подобное, скорее всего, ему стоит отдавать приоритет.

Проблема стандартного WeightedPathSelector из ядра USVM в том, что состояния с одинаковыми весами никак не ранжируются. В случае признака «количество покрытых инструкций» это часто откатывает нас к случайному выбору состояний. Поэтому был реализован WeightedPyPathSelector, который группирует состояния в произвольные UPathSelector на основе некоторого признака, задающегося целым числом. Со внутренним UPathSelector здесь тоже можно экспериментировать.

3.6.2 Выбор действия внутри операции peek

На данный момент был реализован базовый алгоритм, работающий по следующей схеме. Всего были выделены 4 варианта действий:

1. Выбрать состояние из очереди без отложенных ветвлений.
2. Сгенерировать состояние с конкретизированным типом из отложенного ветвления, для которого конкретный тип не был подобран ранее.
3. Выбрать состояние из очереди с отложенными ветвлениями.
4. Сгенерировать состояние с конкретизированным типом из отложенного ветвления, для которого ранее какой-то конкретный тип уже был подобран.

При этом в действиях 2 и 4 необходимо выбрать отложенное ветвление. Можно это делать случайно, а можно пытаться реализовать стратегию,

подобную RandomTreePathSelector. В полной мере вторая идея реализована не была, был пока опробован более простой вариант. Отложенные ветвления группируются по последней инструкции. В момент выбора сначала равномерно выбирается инструкция, и только потом состояние. Такой подход призван компенсировать дисбаланс, который возникает, когда на каждой итерации цикла на одной и той же инструкции возникает отложенное ветвление по типу элемента массива. Правда более правильное решение проблемы типов содержимого массива, вероятно, заключается в полноценной поддержке дженериков, что сейчас есть в планах.

Каждое из перечисленных вариантов действий выполняется с некоторой вероятностью. Сейчас они были подобраны произвольным образом. Чтобы добиться оптимальных значений вероятностей, в будущем стоит попробовать методы обучения с подкреплением. Вероятно, стоит также учитывать, что в каждый момент времени не все 4 варианта могут быть доступны, и в разных конфигурациях набор вероятностей может быть разным.

3.6.3 Выбор рейтинга типов

Самая простая стратегия для выбора рейтинга типов — брать случайный или перебирать все по очереди. Второй вариант используется как базовое решение.

Чуть более сложный способ — пытаться ранжировать рейтинги на основании того, насколько состояние-разведыватель было информировано, когда генерировало его. Простой способ это сделать — посчитать количество и размеры протоколов, которые были сгенерированы для приоритизации типов. Такой вариант был реализован, и он действительно дает небольшой прирост покрытия.

На данный момент кажется, что наиболее перспективное направление для выбора рейтинга типов — это учет покрытия. Сейчас общая инфрастуктура для учета статистики в машине для языка Python развита недостаточно хорошо, поэтому пока это только планы.

Здесь идея состоит в том, что разные ветви исполнения действительно могут соответствовать разным типам. Состояние-разведыватель исследует

одну конкретную ветку для одного конкретного типа и собирает для него подсказки. Соответственно, в вершине дерева отложенных ветвлений необходимо хранить, какие рейтинги соответствуют каким веткам, и какие ветки мы уже смогли покрыть исполнениями с конкретными типами. Приоритет должен отдаваться тем рейтингам, которые потенциально смогут помочь покрыть инструкции, до сих пор покрытые только имитирующими объектами.

3.6.4 Планы на будущее

Вопрос о том, какие эвристики наиболее удачны, пока остается открытым. Из-за того, что PyVirtualPathSelector предполагает вариативность в довольно большом наборе аспектов, количество возможных конкретных конфигураций очень велико. В такой ситуации легко попасть в классическую ловушку «переобучения»: если проверить на одних данных эффективность множества разных стратегий, то с большой вероятностью какая-то из них случайно будет показывать лучшие результаты. Для избежания данной проблемы в будущем необходимо разработать механизм валидации, но это все выходит за рамки данной работы. Здесь для оптимального решения стоит привлекать методы математической статистики и машинного обучения.

4. Полученные результаты

Реализованная система символьного исполнения для языка Python была встроена в инструмент генерации юнит-тестов UnitTestBot [26], [27]. Было проведено сравнение полученного инструмента с другими генераторами тестов для Python.

4.1. Соревнование SBFT 2024

UTBot Python, в котором комбинировались наш символьный движок и фаззер, участвовал в международном соревновании SBFT 2024 и занял второе место после фаззера Pynguin [28]. Также в соревновании участвовали Klara и Hypothesis Ghostwriter.

К сожалению, файлы, на которых проводилось сравнение инструментов на соревновании, выложены не были, поэтому результаты не воспроизводимы.

4.2. Детали проведения наших экспериментов

Для проведения наших собственных экспериментов была использована инфраструктура с соревнования SBFT 2024 [29].

Свои эксперименты мы проводили на тестовых данных[4], которые основывались на одном публичном репозитории, содержащем реализации алгоритмов на языке Python [30]: сортировки, структуры данных, алгоритмы на графах. В инфраструктуре с соревнования есть определенные проблемы, из-за чего сейчас на ее основе проводить замеры на реальных проектах со внешними зависимостями достаточно проблематично.

Исходно в соревновании замеры проводились по трем метрикам: покрытие по строкам, покрытие по веткам и значения мутационного тестирования. Мутационное тестирование занимает много времени, поэтому было решено на своих экспериментах данную метрику не использовать. Во всех последующих таблицах и графиках указано покрытие по строкам.

В качестве лимита времени во всех экспериментах устанавливалось 240 секунд на один файл. В нашем тестовом наборе всего 117 файлов.

[4] `https://github.com/tochilinak/python_benchmarks`

Как было указано в разделе 3.5, символьное исполнение можно запускать с разными вариантами алгоритма выбора следующего символьного состояния. Для данных экспериментов был выбран вариант, где в качестве UPathSelector был выбран WeightedPyPathSelector (описанный в разделе 3.6.1), учитывающий количество инструкций без повторов, со внутренней очередью RandomTreePathSelector. Также включена эвристика для приоритизации рейтингов типов (описанная в разделе 3.6.3). По сравнению с самой базовой реализацией прирост составил 2%.

Вариант PyVirtualPathSelector	Покрытие строк
Базовый	0.791
С эвристиками	0.811

Таблица 3: Покрытие символьного исполнения с разными вариантами PyVirtualPathSelector.

4.3. Сравнение с CrossHair

Разработчики CrossHair создали репозиторий [31], который интегрирует данный инструмент в инфраструктуру с соревнования. На основе этого репозитория мы провели собственное сравнение нашей символьной машины с CrossHair.

Как было отмечено в главе 1.1, кодогенерация в CrossHair реализована недостаточно хорошо, в частности, плохо генерируются блоки с assert. Чтобы при сравнении движков символьного исполнения данная проблема не влияла на результаты, мы упростили кодогенерацию в CrossHair. Версия, которая была использована для проведения экспериментов, доступна в публичном репозитории[5].

Для данного сравнения инструмент UTBot Python запускался с выключенным фаззингом, данные генерировала только символьная машина.

В таблице 4 приведено итоговое покрытие, достигнутое нашим инструментом и CrossHair.

На рис. 5 представлен график покрытия строк по файлам. Каждая строка соответствует одному файлу. Серым показано количество строк в файле,

[5] `https://github.com/tochilinak/python-tool-competition-2024-crosshair`

Инструмент	Покрытие строк
utbot-python	**0.811**
CrossHair	0.565

Таблица 4: Сравнение CrossHair с нашей символьной машиной.

зеленым — количество строк, покрытых нашим инструментом, красным — CrossHair. Видно, что CrossHair работает менее стабильно, чем наш инструмент: на довольно большом количестве файлов он завершил работу с ошибкой и ничего не сгенерировал.

4.4. Сравнение с фаззингом в UTBot Python

Как было сказано в самом начале данной работы, изначально задача создания символьного анализатора для языка Python возникла с целью улучшить существующий инструмент для генерации юнит-тестов, в котором уже реализован фаззинг. Чтобы понять, получилось ли этого достичь, была проведена генерация тестов на подготовленных данных в трех режимах: только фаззинг, только символьное исполнение и оба метода, работающие параллельно.

Результаты замеров приведены в таблице 5. Распределение покрытия по файлам приведено на рис. 6. Зеленым обозначено покрытие при запуске только символьного исполнения, красным — только фаззинга. Видно, что иногда лучше работает один метод, иногда — другой, поэтому они друг друга дополняют, и при параллельном использовании обоих методов результат получается наилучшим.

Метод генерации	Покрытие строк
Символьное исполнение	0.811
Фаззинг	0.796
Символьное исполнение + фаззинг	**0.879**

Таблица 5: Сравнение символьного исполнения и фаззинга.

4.5. Сравнение с Pynguin

Инструмент Pynguin можно конфигурировать большим количеством разных способов. Для эксперимента была взята версия с соревнования [32].

Единственный измененный параметр — лимит времени, который был уменьшен с 350 до 240.

Для этого сравнения UTBot Python запускался с символьным исполнением и фаззингом, работающими параллельно. Результаты представлены в таблице 6 и на рис. 7. На графике зеленым отображается наш инструмент, красным — Pynguin.

Инструмент	Покрытие строк
utbot	**0.879**
pynguin	0.769

Таблица 6: Сравнение UTBot Python и Pynguin.

Как выяснилось, инструмент Pynguin довольно нестабильный: он довольно часто зависал или завершал работу с какой-нибудь внутренней ошибкой, что видно по рис. 7. Ниже представлен пример ошибки, с которой заканчивался анализ на большом количестве файлов:

```
python_benchmarks/graphs/dijkstra_algorithm.py
failed with FailureReason.UNEXPECTED_ERROR
- 'types.UnionType' object has no attribute '__qualname__'
```

Тем не менее, видно, что все же есть достаточно примеров, на которых Pynguin действительно довольно сильно обгоняет наш инструмент по покрытию.

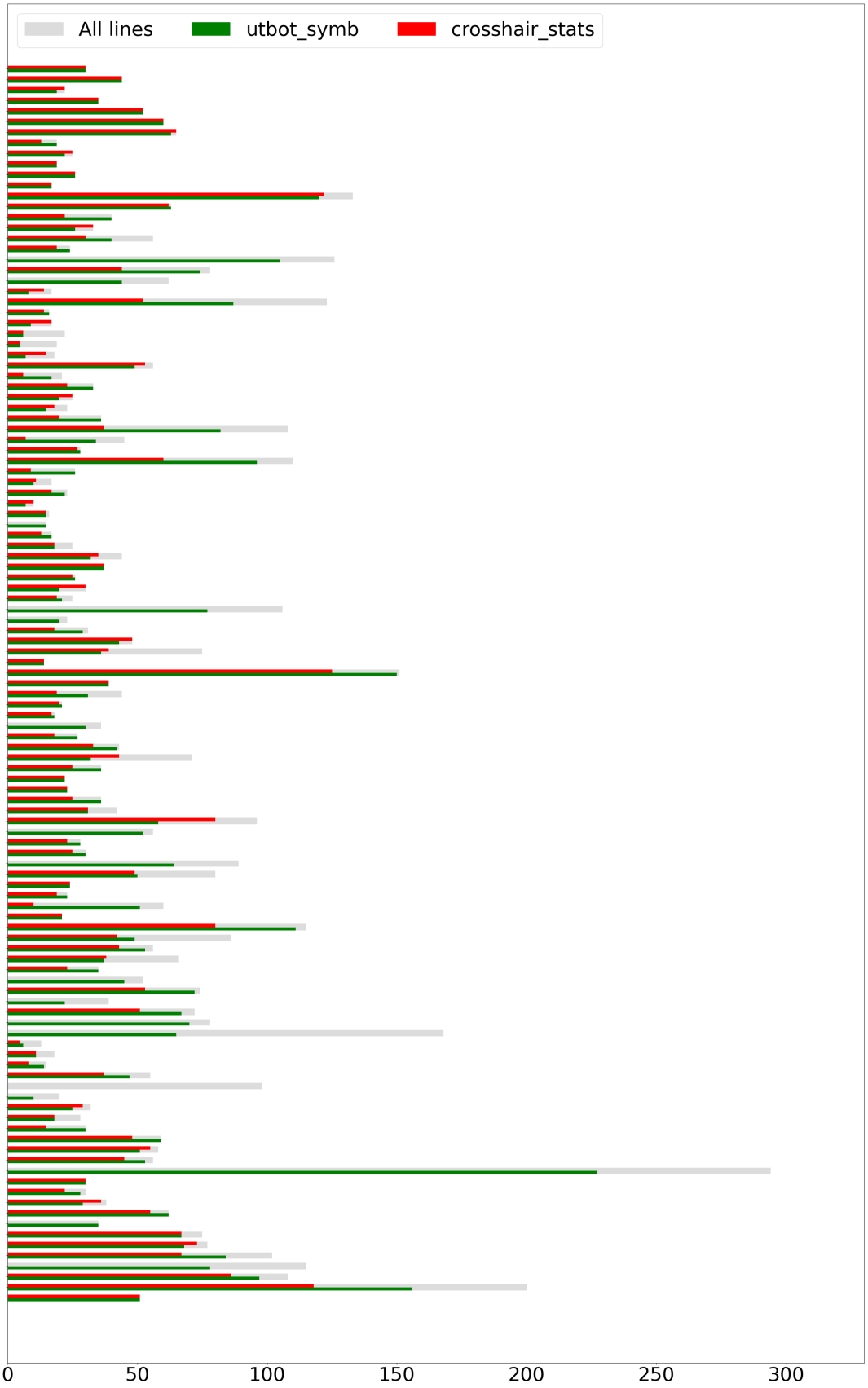


Рис. 5: Сравнение CrossHair с нашей символьной машиной по файлам.

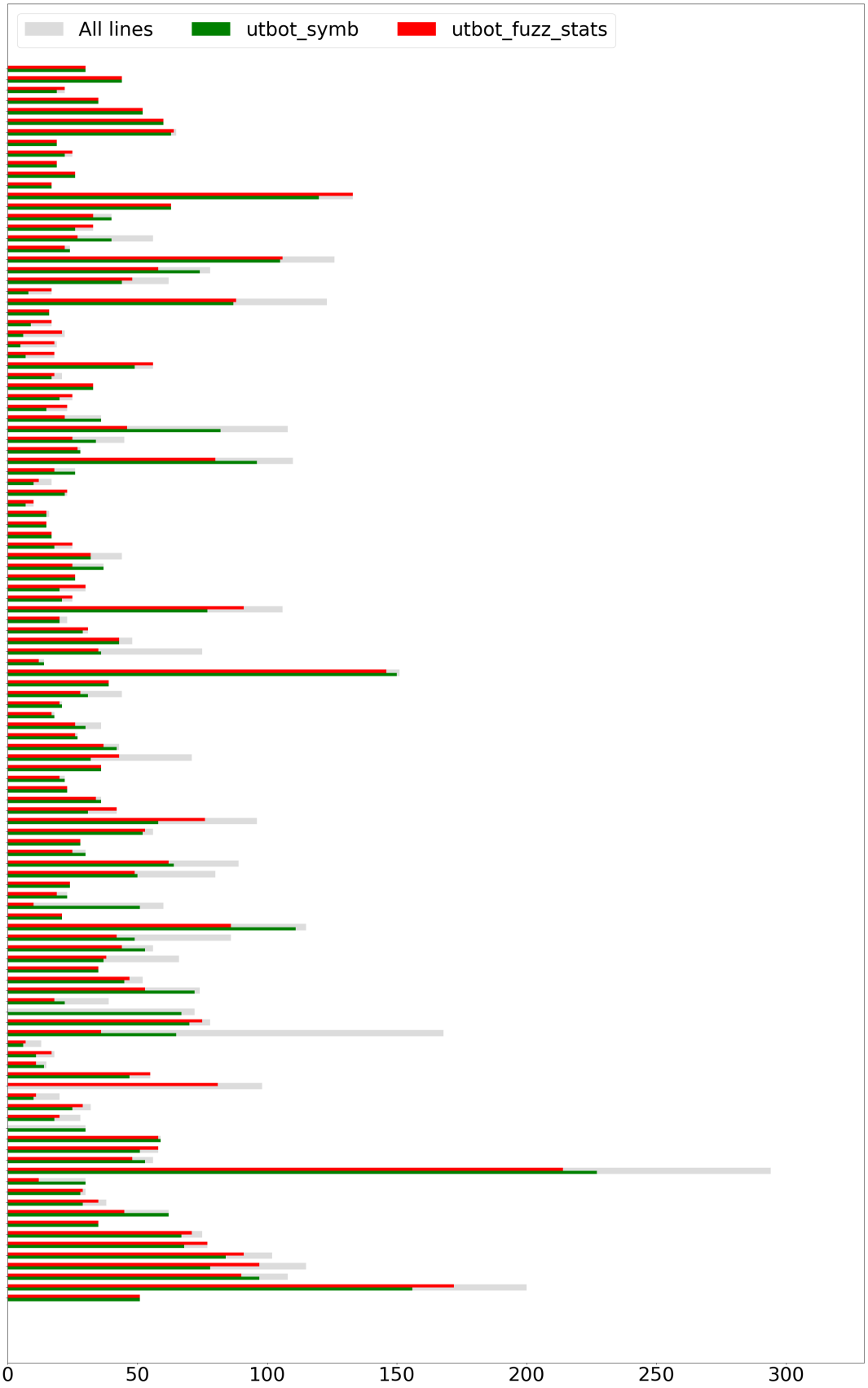


Рис. 6: Сравнение фаззинга и символьного исполнения по файлам.

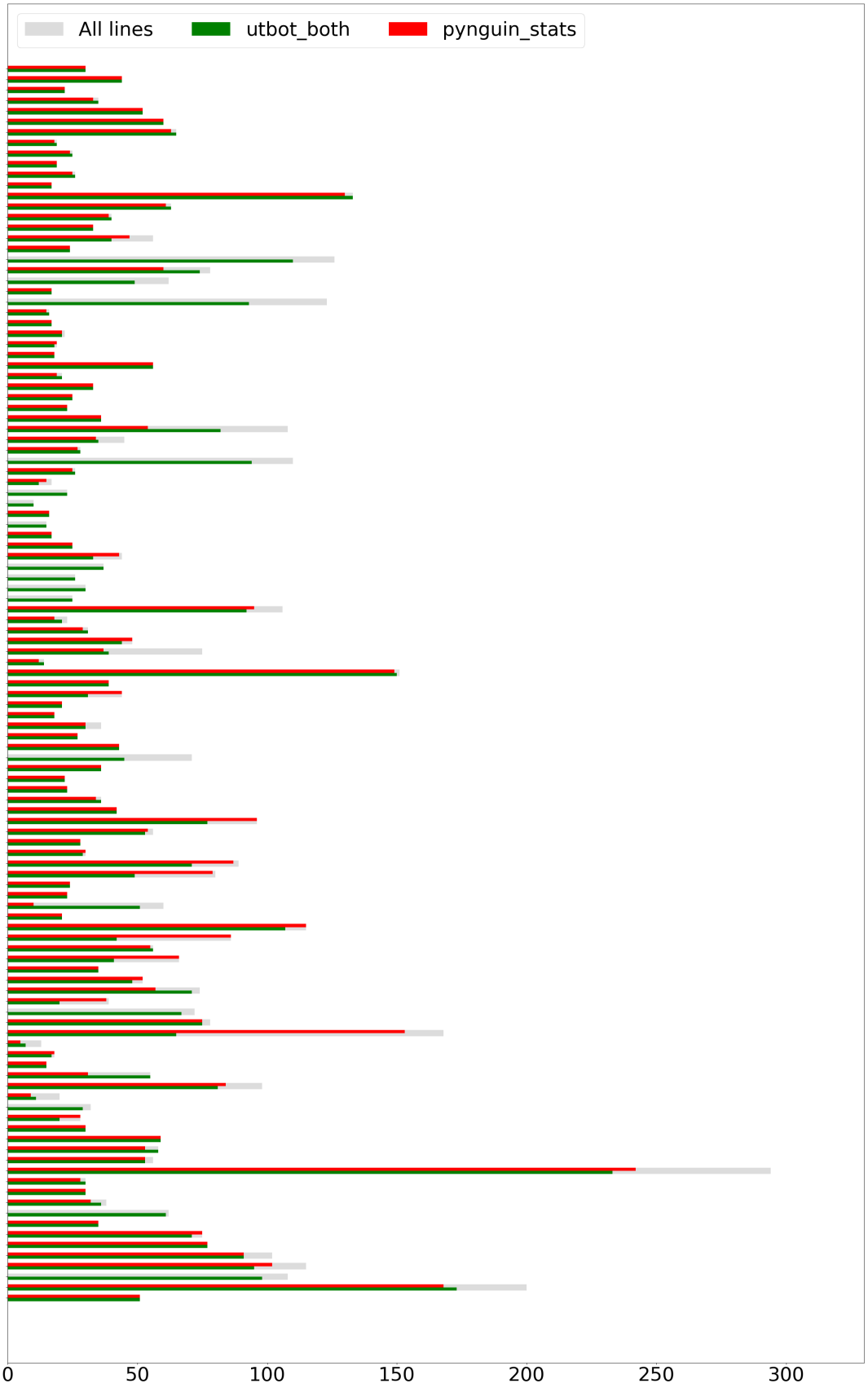


Рис. 7: Сравнение UTBot Python и Pynguin по файлам.

Заключение

В результате данной работы были успешно выполнены следующие задачи:

1. Были проанализированы существующие реализации символьного исполнения для языка Python. Их немного, и в лучшей существующей реализации (инструмент CrossHair) имеется ряд проблем. В частности, инструмент CrossHair недостаточно хорошо подбирает типы в тех ситуациях, когда они не указаны в виде аннотаций.

2. Были разработаны подходы для решения проблем символьного исполнения, специфичных для языка Python. Для облегчения проблемы большого количества инструкций было решено комбинировать символьный интерпретатор с конкретным. Для облегчения проблемы нативных функций было решено модифицировать исходный код CPython. Для решения проблемы неизвестных типов был разработан подход с имитирующими объектами и отложенными ветвлениями.

3. Данные подходы были реализованы в рамках машины USVM. Основная часть кода была написана на Kotlin, но также присутствуют модули, написанные на C и на Python.

4. Было проведено сравнение полученного инструмента с существующими аналогами. Во-первых, инструмент UTBot Python с встроенной символьной машиной стал участником международного соревнования SBFT 2024, где занял призовое место. Во-вторых, был проведен эксперимент по генерации тестов с помощью различных инструментов для реализаций алгоритмов. Реализованная в данной работе символьная машина набрала на 24% большее покрытие, чем наиболее полная существующая реализация символьного исполнения для Python — CrossHair.

Реализация доступна в публичном репозитории:
`https://github.com/UnitTestBot/usvm/tree/tochilinak/python`.

Список литературы

[1] Nidhra Srinivas, Dondeti Jagruthi. Black Box and White Box Testing Techniques - A Literature Review // International Journal of Embedded Systems and Applications (IJESA). — 2012. — Vol. 2, no. 2. — P. 29–50. Algorithms, 12:3, 2016.

[2] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey., ACM Computing Surveys, 2019.

[3] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.

[4] CrossHair. URL: `https://github.com/pschanely/CrossHair`

[5] A.M. Bruni, T. Disney, C. Flanagan. A Peer Architecture for Lightweight Symbolic Execution, 2011. URL: `https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf`

[6] PyExZ3. URL: `https://github.com/thomasjball/PyExZ3`

[7] PySym. URL: `https://github.com/bannsec/pySym`

[8] Pynguin. URL: `https://github.com/se2p/pynguin`

[9] EvoSuite. URL: `https://github.com/EvoSuite/evosuite`

[10] G. Jahangirova and V. Terragni, "SBFT Tool Competition 2023 - Java Test Case Generation Track,"2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), Melbourne, Australia, 2023, pp. 61-64, doi: 10.1109/SBFT59156.2023.00025.

[11] Klara. URL: `https://github.com/usagitoneko97/klara`

[12] Ограничения Klara. URL: `https://klara-py.readthedocs.io/en/latest/limitation.html`

[13] Hypothesis. URL: https://github.com/HypothesisWorks/hypothesis

[14] Hypothesis Ghostwriter. URL: https://hypothesis.readthedocs.io/en/latest/ghostwriter.html

[15] Альтернативные реализации языка Python. URL: https://wiki.python.org/moin/PythonImplementations

[16] PyPy. URL: https://github.com/pypy/pypy

[17] Jython. URL: https://github.com/jython/jython

[18] IronPython 3. URL: https://github.com/IronLanguages/ironpython3

[19] USVM. URL: https://github.com/UnitTestBot/usvm

[20] KSMT. URL: https://github.com/UnitTestBot/ksmt

[21] Python C API. URL: https://docs.python.org/3/c-api/index.html

[22] Документация по PyTypeObject. URL: https://docs.python.org/3/c-api/typeobj.html

[23] PEP 483 - The Theory of Type Hints. URL: https://peps.python.org/pep-0483/

[24] PEP 484 - Type Hints. URL: https://peps.python.org/pep-0484/

[25] Mypy. URL: https://github.com/python/mypy

[26] UTBotJava. URL: https://github.com/UnitTestBot/UTBotJava

[27] UTBot Python for SBFT 2024. URL: https://github.com/UnitTestBot/UTBotPythonSBFT2024

[28] Nicolas Erni, Al-Ameen Mohammed Ali Mohammed, Christian Birchler, Pouria Derakhshanfar, Stephan Lukasczyk, and Sebastiano Panichella. SBFT Tool Competition 2024 - Python Test Case Generation Track. URL: https://arxiv.org/pdf/2401.15189

[29] SBFT Python Tool Competition 2024 Infrastructure. URL: https://github.com/ThunderKey/python-tool-competition-2024

[30] The Algorithms. Python. URL: https://github.com/TheAlgorithms/Python

[31] CrossHair for SBFT 2024. URL: https://github.com/pschanely/python-tool-competition-2024-crosshair

[32] Pynguin for SBFT 2024. URL: https://github.com/ThunderKey/python-tool-competition-2024-pynguin